Trellis
Object-Based Environment
Language Reference Manual

DEC-TR-372

Craig Schaffert
Topher Cooper
Carrie Wilpolt

Version 1.1
November 25, 1985

Eastern Research Lab
Digital Equipment Corporation
Hudson, Massachusetts

© Digital Equipment Corporation 1985
Any Unauthorized Reproduction is Strictly Prohibited
All Rights Reserved

Preface

Trellis[1] is an object-based programming environment. It consists of a compiler and a set of program development tools which support editing, debugging, and browsing. The tools share a common multiple-window, mouse-driven user interface. The Trellis environment runs under VMS on a VAXstation II.

The object-based programming language was developed within Digital and was initially called **Owl**. It will be referred to by that name within this document. Owl combines object-based programming along with strong typing to support the construction of large complex software systems.

This report is a manual for Version 1 of the Owl language. It is intended for Owl programmers. To learn about Owl, you should read the Owl Tutorial first.

The following researchers have worked on the design and implementation of Trellis and Owl: Toby Bloom, Bruce Bullis, Wayne Citrin, Topher Cooper, Dan Halbert, Mike Kilian, Ken King, Walt Kohler, Pete Lee, Eliot Moss, Pat O'Brien, Susan Rajunas, Craig Schaffert, Ray Shapiro, and Carrie Wilpolt.

---

[1] Trellis is a trademark of Digital Equipment Corporation.

Contents

1 Introduction 1

2 Concepts 3

2.1 Objects and Operations 3

2.2 Types 6

2.3 Type Safety 10

2.4 Program Structure 11

3 Syntax 13

3.1 Identifiers 14

3.2 Reserved Words 14

3.3 Labels 14

3.4 Integer Literals 15

3.5 Real Literals 15

3.6 String Literals 15

3.7 Comments 16

4 Simple Expressions 17

4.1 Value Required Contexts and Type Correctness 18

4.2 Order of Evaluation 18

4.3 Literals 19

4.4 Identifiers 19

4.5 Type Specs 19

4.6 Invocations 20

4.7 Unary Operation Expressions 21

4.8 Binary Operation Expressions 21
4.9 Record Constructors 23
4.10 Sequence Constructors 24
4.11 Simple Component Selection 25
4.12 Indexed Component Selection 25
4.13 Assignments 25
4.13.1 Variable Assignment 26
4.13.2 Simple Component Assignment 26
4.13.3 Indexed Component Assignment 26
4.14 Blocks 27
4.15 Precedence 27

5 Control Expressions **29**
5.1 Bodies 29
5.2 Variable Declarations 30
5.3 IF Expressions 31
5.4 CASE Expressions 31
5.5 TYPE_CASE Expressions 32
5.6 LOOP Expressions 33
5.7 FOR Expressions 34
5.8 EXIT Expressions 35
5.9 CONTINUE Expressions 35
5.10 RETURN Expressions 35
5.11 YIELD Expressions 36
5.12 Blocks 36
5.13 LEAVE Expressions 37
5.14 SIGNAL and RESIGNAL Expressions 37
5.15 Exception Handlers 38

6 Types and Operations **41**

6.1 Type Definitions 41
6.2 Operation Definitions 42
6.3 Component Definitions 44
6.4 Object Creation 47
6.5 Visibility 48
6.6 Subtyping and Inheritance 48
6.6.1 Subtype Compatibility 49
6.6.2 Inheritance of Specifications 50
6.6.3 Inheritance of Implementations 51
6.7 Fixed Names 52
6.8 Other Attributes 53

7 More on Types **55**

7.1 Type Generators 55
7.1.1 Using Parameters 57
7.1.2 Subtyping 57
7.1.3 Fixed Names 58
7.2 Special Type-Specs 59
7.3 When Clauses 60
7.4 Union Types 62

A Syntax Summary **65**

A.1 Lexemes 65
A.2 Simple Expressions 66
A.3 Control Expressions 67
A.4 Definitions 69

B Contracted Forms **73**

B.1 Unary Operators 73
B.2 Binary Operators 73
B.3 Components 74
B.4 Constructors 74
B.5 Precedence 74

C Special Names 77

Index 81

Chapter 1

Introduction

The Owl programming language is intended to be the primary language used in the Object-Based Systems project. As such it has three major objectives:

- Demonstrate the utility of the object-based style of programming;
- Provide sufficient features to support
  - all applications programming,
  - all database programming (with suitable library support),
  - all of the Programming Environment; and
- Investigate the advantages and disadvantages of
  - type hierarchies (with multiple supertypes),
  - implementation inheritance (with multiple sources),
  - "unary" operations,
  - integrating the language and database.

Owl is not intended to directly provide a database query system, window package, or any other application specific facility. It is designed to be used in constructing such facilities for users. Our major goal has been to support the long-term maintenance and evolution of programs, rather than to provide a rapid prototyping system.

Owl is still in its formative stages. In several instances language decisions were made on the basis of "try it and see." Thus this manual does not describe a polished language, but an evolving tool for experimentation and discussion.

Language extensions are expected in the following areas: predicate quoting mechanisms (for database work), multiple activities, synchronization, transaction support (for error recovery in distributed systems), and name-space control (for programming-in-the-large).

This document is intended to be a reference manual for Owl programmers. To learn about Owl, you should read the Owl Tutorial first. That will give you a good understanding of object-based programming, type hierarchies, and how to use Owl.

Owl is part of an integrated programming environment. Owl programming is viewed as modifying and extending an ever-growing database of type and operation definitions. There will be many tools – some explicit, others implicit – for evolving this database, checking it for consistency, and extracting information from it. The Owl Programming Environment Manual describes the current state of these tools. The Owl Library Manual documents the most useful of the many definitions currently available.

The rest of this manual is structured as follows:

CHAPTERS

Concepts	A discussion of the basic concepts of Owl.
Syntax	An overview of the syntax, including identifiers, literals, comments, and so on.
Simple Expressions	Various ways to compute new objects or manipulate old ones.
Control Expressions	Conditionals, loops, exceptions, and so on.
Types and Operations	How to define (and implement) a new type.
More on Types	Defining families of types, and more advanced topics.

APPENDICES

Syntax Summary	All the syntax rules (for quick reference).
Contracted Forms	Unary and binary operators, component selection, etc.
Special Names	Reserved words and special identifiers.

INDEX

Chapter 2

Concepts

This chapter introduces the basic concepts of Owl, concentrating on the differences between Owl and more traditional languages. We will discuss objects, and how they are manipulated by operations. We will discuss types, which define the behavior of objects. Finally we will describe the overall structure of Owl programs. You should have already read the Owl Tutorial.

2.1 Objects and Operations

This section describes four ideas that are central to understanding Owl: objects, operations, variables, and assignment.

In designing a program, it is generally more fruitful to concentrate on the data to be manipulated rather than the code that does the manipulation. This insight has given rise to the *object-based* style of programming. In this approach, the data object (and its logical behavior) is the central organizing principal. Algorithmic structure is secondary (although still important).

Objects

An *object* in Owl is any piece of data. Objects commonly model real-world entities, such as a person, a document, or an i/o device. Objects also model abstract things, like a queue, a matrix, or a filename. Objects are manipulated by applying *operations* to them. Operations can extract information from an object, change the object, or create new objects. (Operations in Owl are similar to procedures or functions in other languages.)

Each object is viewed as a black box. A programmer using an object thinks only in terms of the operations defined for that object, not in terms of the object's representation.

Each object has a *type*. The type of an object determines the operations that can be applied to it. Since objects are only manipulated by operations, an object's type completely defines (and controls) its behavior.

Objects can range from the small and simple, to the large and complex. The booleans `true` and `false` are two of the simplest objects – they have no internal structure, and very few operations. At the other extreme, a document object might be quite complex – full of chapters, sections, paragraphs, words, illustrations, tables, internal cross references, and so on.

Complex structures are frequently built up out of many objects that *refer* to each other. Consider a directed graph. Each node in the graph could be an object, and there would be operations, like `follow_first_arc`, that take us from one node to another. If the `follow_first_arc` operation on node A returns Z, we say that A refers to Z. Another node B could also refer to Z. We then say that A and B *share* Z. The sharing relations between objects can be made as sophisticated as desired. We rarely say that one object *contains* another, because this erroneously implies that sharing is impossible.

Objects fall into two broad categories: *constant* objects, and *mutable* objects. Constant objects have no time-varying properties – they are pure values. Examples of constant objects include integers (like `5`) and strings (like `"Henry"`). Constant objects can be structured, but that structure cannot be changed. Mutable objects have time-varying properties. For example, a display screen object may have a "cursor position" which can be modified by display operations. A graph object may have operations that add and delete nodes and arcs. Both graphs and display screens would thus be mutable objects.

"Constant" and "mutable" are intrinsic properties of an object. An object is constant if none of its operations permit it to be modified. The distinction between these two kinds of objects is important when objects are shared. Sharing of mutable objects allows communication through the shared object. Sharing of constant objects is not detectable because constant objects cannot be modified.

Operations

In Owl, operations are invoked in the familiar way. For example, the expression

```
display (the_drawing, the_screen, the_position)
```

invokes the display operation on an object called `the_drawing`. Two additional arguments, `the_screen`, and `the_position`, indicate where the drawing is to be displayed.

In general, an operation invocation specifies an operation name and one or more arguments. The first argument is called the *controlling argument* because it determines the actual code to be executed. An operation can examine or modify any of its arguments, or any other objects that can be reached via the arguments. An operation can also return a result

object. Typically, all communication between the operation and its caller is embodied in these objects.

Consider the possibility that different kinds of drawings might need different display algorithms. Certainly, displaying an integer would be (internally) different from displaying a house. However the abstract notion of "display" is the same in both cases. By having the controlling argument select the actual code executed, we can write a single piece of code that uses the display operation, and then apply this code to different kinds of displayable objects. The appropriate display algorithm will be called for each (different) type of displayable object.

This capability of Owl is more than just a flexible naming scheme, particularly when combined with the notion of subtypes. As we shall see later, it contributes considerably to the clarity and extensibility of Owl programs.

A common programming problem is how to deal with errors. Or, more generally, what do we do when a piece of code cannot achieve its goals? For example, assume that the `first` operation on lists is supposed to find the first element of a given list. What does it do if the list is empty?

When an Owl operation cannot achieve its goal, it *signals* an *exception.* An exception is an alternate (named) termination state for an operation. The `first` operation on lists could signal `Empty_List`, and an input operation on files could signal `End_of_File`. An operation's caller typically handles exceptions separately from normal returns. This separation leads to clearer code, and better documentation. Each exception that an operation can signal can be accompanied by a return object, just as in the normal return case. (Exceptions are described further in Sections 5.14 and 5.15.)

Another common programming situation is the need to do some work for each element of a collection. We want to print a salary check for each employee, or execute each command in a file. Owl provides a special kind of operation – called an *iterator* – that makes this particularly easy to do. Iterators are closer to coroutines than to conventional procedures, since they produce a series of results rather than just one. The `for` loop

```
for p: Professor in faculty (our_department) do
    grant_tenure (p);
end for;
```

grants tenure to each professor in `our_department`. `Faculty` is an iterator that *yields* each professor in the department in sequence. (Iterators are described further in Sections 5.7, 5.11, and 6.2.)

Variables and Assignment

When we write programs, we need names to denote objects. In the above `for` loop, `p` denotes (in sequence) each of the professors in `our_department`. A local name for an

object is called a *variable*. A variable declared in the body of an operation is local to a single invocation of that operation. In fact, the variable `p` is local to the body of the `for` loop. We say that a variable *denotes* an object rather than containing it, because more than one variable can denote the same object at once.

Variables are not objects, they just name (denote) objects. Assignment causes a variable to denote another object. Thus, `x := y` causes `x` to denote the same object as `y` without modifying the state of any object.

Assignment is an important part of operation invocation. The body of an operation declares a local variable for each argument that it expects, and these locals are initialized with the actual argument objects when the operation is invoked. Thus the argument objects are shared (and can be modified), but the variables that the caller used to name them are not shared. In fact, an operation has no way to access or modify its caller's local variables.

2.2 Types

Types are the major organizing principle in Owl. In this section we describe how types are specified and implemented. We also introduce the notions of *subtyping*, *inheritance*, and *type generators*.

Basic Ideas

Every object has a type, and this type defines the object's behavior. All the objects of a type can have the same operations applied to them. Some example types are:

`Integer` — Integer objects include `0`, `1`, `-1`, `2`, `-2`, and so on. The operations applicable to integers include addition, subtraction, equality, the ordering operations, conversions to and from decimal form, and many others.

`String` — Example strings include `""`, `"x"`, `"Patrick Henry"`, and `"PATRICK HENRY"`. The string operations include length, concatenation, extracting individual characters and substrings, ordering operations, and an iterator, `elements`, which yields each of the string's characters one character at a time.

`Set_of_Professors` — Any finite group of professors, including {}, {`Russell`, `Whitehead`}, or `The_Faculty_of_Oxford`, could form an object of this type. Operations might include set union, intersection, a subset test, and an elements iterator. We might also have average salary, or maximum age operations.

`Text_Document`	With this type, we could create as many text documents as needed, add text to them, extract sections or chapters as subdocuments, change fonts, or prepare an index. Text documents could behave in whatever way is needed for the application at hand.
`Display_Screen`	A display screen is an idealized i/o device. It provides enough operations (e.g. `write_string`, `draw_figure`) so we can conveniently implement the display operations needed for other types (e.g. `Integer`, `Text_Document`, `Pert_Chart`).

Operations defined as part of a type `T` exist to manipulate the objects of that type. Typically, such an operation will have an object of type `T` as its first (controlling) argument. However, not all `T` operations manipulate existing `T` objects. Creation operations are the best example of this, because they produce new `T` objects, rather than examining or modifying ones we already have. Operations that manipulate an existing object are called *instance* operations. Operations that create new objects or otherwise affect the type as a whole are called *class* operations.

Think of the type `Symbol_Table`, which associates values with identifiers. A program that uses symbol tables might contain the following code:

```
symbols := create (Symbol_Table);
insert (symbols, new_identifier, its_value);
```

`Create` and `insert` are both symbol table operations. `Create` is a class operation with controlling argument `Symbol_Table`. `Insert` is an instance operation with a particular symbol table (the one denoted by `symbols`) as its controlling argument.

As this example shows, types are also objects in Owl. Type objects have the type `Type`. The class operations on a type `T` are only applicable to the object `T` itself. However, there are also a few general instance operations applicable to all type objects (see the Owl Library Manual).

There are two views of any type: the *specification*, and the *implementation*. The specification is an external view of the type. It describes the behavior of a type, but does not say how that behavior is realized. The implementation is an internal view of the type. It tells how the specified behavior is actually achieved.

A full specification of a type must characterize each possible object of that type, and define the full behavior of each of the operations. Owl provides notation to describe various attributes of the objects (including whether they are constant or mutable), and to describe each of the operations. Operation descriptions include the number and types of arguments, results, exceptions, and so on. Owl also provides means for attaching further information to a type definition, but such information will be handled by tools other than the compiler (see the Programming Environment Manual).

Owl does not provide a complete notation for writing specifications. For example, there is no notation in Owl for stating that a `sort` routine is stable, or that it takes a certain amount of space or time. The part of an operation's specification that can be written in Owl is called the operation's *interface*. In this manual, "specification" means full specification. When we want to restrict our attention to that part of the specification that has special notation in Owl, we will use the word "interface".

After a type is specified, it must be implemented. A type implementation describes how objects of that type are represented, and how each operation is actually implemented in terms of that representation.

Internally, each object is represented as a collection of *fields*. Each field contains a reference to some other object. The fields of an object are not directly accessible to users of that object. However, the operations defined on that object can read and write the object's fields.

The fact that only operations defined on type `T` can directly manipulate the representation of `T` objects has three benefits. First, since only the implementer of `T` can access the fields, he can insure that `T` objects behave exactly as defined. Second, the implementation of `T` can be verified or tested in isolation from the code that uses it. Third, the implementation of `T` can be tuned or even changed radically without affecting the users of the type.

Sometimes, implementers of a type find it convenient to implement one or more operations in terms of other operations not listed in the specification. Such internal implementation-specific operations are called *private* operations, in contrast to the external *public* operations we have been discussing up to now. Private operations are not available to users of the type.

Subtypes and Inheritance

A type is a description of object behavior. Some descriptions are more general than others. For example, the type `Displayable_Object` stipulates that its objects must have a `display` operation:

```
operation display (me, on: Display_Screen, where: Position)
```

which draws the object on the screen at the given position. This is a very high-level description. It is satisfied by many types of objects, including integers, strings, drawings, text documents, and so on. To capture the notion that every string is also a displayable object, we say that `String` is a *subtype* of `Displayable_Object`. Conversely, we say that `Displayable_Object` is a *supertype* of `String`.

A type definer must explicitly declare that a type `S` is to be considered a subtype of another type `T`. This declaration asserts that every object of type `S` is also logically of type `T`. Any property that holds for `T` objects will also hold for `S` objects. Operations defined on `T` objects will be defined on `S` objects (and satisfy the same specification).

Consider the string `"Patrick Henry"`. This object has type `String`, but it also has type `Displayable_Object`, since `String` is a subtype of `Displayable_Object`. The most specific type of an object – the type that fully defines that object's behavior – is called the *base type* of the object. For example, the base type of `"Patrick Henry"` is `String` rather than `Displayable_Object`.

Every object has a unique base type. This is established when the object is created, and never changes. However, not all types are base types. `Displayable_Object` embodies the notion of "displayability" independently of any specific kind of displayable object. All displayable objects are actually integers, strings, text documents, or some other subtype of `Displayable_Object`. We call a high-level type like `Displayable_Object` an *abstract* type. Abstract types specify a behavior that is common to several other types, but do not introduce any new objects. Types that do introduce new objects are called *base* types.

A type can be a subtype of more than one type. The type `String` is also a subtype of the type `Collection`. A collection is an object with an **`elements`** iterator that yields each of the members of the collection. Strings can be declared to be collections (of characters) because the **`elements`** iterator on strings yields each of the characters in the string.

The subtype/supertype relations form a partial order on all types. The single type `Object` is automatically a supertype of all types, and includes all objects. There are a small number of operations that are defined on type `Object` (and are thus applicable to all objects). See the Owl Library Manual for details.

If `S` is declared to be a subtype of `T`, then `S` *inherits* the specifications and implementations of `T`'s operations. Only additional or changed operations need be written for `S`. Defining `S` in terms of its differences from `T` leads to shorter definitions, and code that is easier to understand and maintain. Inheritance is a very powerful concept and discussed fully in Section 6.6.

We have now introduced all the important ideas about types except one – the *type generator*. A type generator is a generalization of a simple type. It defines a family of related types. For example, `List` is a type generator that produces a family of list types. `List[Integer]` is a list-of-integers type, `List[String]` is a list-of-strings type, and `List[List[Displayable_Object]]` is a list-of-lists-of-displayable-objects type. Each of these types has the same basic list behavior, but their objects contain different types of elements. By defining the `List` type generator once, we produce an unlimited family of list types. The values in square brackets are called *parameters*, and each member of the family is called a *parameterized type*. Type generators are described in Chapter 7.

A programming language's types are a major source of its power. As more types are defined this power grows. Owl starts out with a rich type library, and this library continually expands as programs are written. Owl's initial type library is described in the Owl Library Manual.

2.3 Type Safety

Owl is a type-safe language. Any attempt to apply an operation to an object that does not have such an operation will be caught at compile time. This prevents certain run time errors, but, more importantly, it improves program structure.

Type safety is achieved by specifying the argument and return types of every operation, and attaching a type to every local variable declaration. These *type specs* limit the range of objects that can be passed to operations, returned from operations, or assigned to variables. With these limits, we can always determine the range of objects that any given expression could produce.

Knowing the type of an object means that we know a fair amount about how it will behave. This knowledge is crucial for understanding and maintaining a program as it evolves. Experience also shows that type safety imposes a discipline on the program development process that eliminates a majority of logical errors before the program even runs.

In its simplest form, a type spec is just the name of a type. The declaration

```
var symbols: Symbol_Table;
```

restricts the variable `symbols` to denote only `Symbol_Table` objects. The declaration

```
var screen_contents: List[Displayable_Object];
```

says that `screen_contents` denotes a list of displayable objects. Another form of type spec is the *union* type. The declaration

```
var image: Integer | String | Text_Document;
```

restricts the variable `image` to denote only integers or strings or text documents. Since each of these types have a display operation, we can write

```
display (image, the_screen, the_position);
```

to display the image. Each of the types `Integer`, `String`, and `Text_Document` is a subtype of the union `Integer | String | Text_Document`, which is itself a subtype of `Displayable_Object` (since each element of the union is a subtype of `Displayable_Object`). Type specs are discussed further in Section 4.5.

2.4 Program Structure

In this final section, we discuss issues of program structure and syntax, including *contracted forms*, *components*, and *fixed names*.

An Owl program consists of a set of related type definitions. Each of these type definitions contains a list of subsidiary definitions. These definitions include operations, components, and fixed names, but *not* further type definitions.

Owl does not make a syntactic distinction between statements and expressions. Control structures that would be statements in other languages can be used in Owl expressions. This is allowed whenever the control structure has a "natural" value, but no attempt is made to provide arbitrary values for all control structures.

Owl provides all the conventional infix and prefix operators. However, these are considered to be *contracted forms* for operation invocations. For example, the expression `x + y` is a shorthand for `plus(x, y)`. This policy means that operators like "+" can be used for user defined types such as matrices or polynomials. All the contracted forms are listed in Appendix B.

Many objects are similar to records or arrays in that they are most conveniently viewed as having *components*. A complex number `C` may have a real component `C.real` and an imaginary component `C.imag`. A text document may be viewed as being (in part) an array of lines, so the third line would be `my_page[3]`. Both simple and indexed components are supported for user defined types. This is done by implementing component access through a pair of operations: one for extracting the value of the component, and one for assigning a new value to the component.

A type definition provides operations, components, and *fixed names*. A fixed name is a global name for some object, usually of the type being defined. For example, `true` and `false` are fixed names for the two boolean values, and they are part of the definition of type `Boolean`. Fixed names are discussed in Section 6.7.

We have now reviewed the basic concepts of Owl. The chapters that follow will build on these ideas and (we hope) make them more precise.

Chapter 3

Syntax

An Owl program is written as a series of *lexemes*. Lexemes include identifiers, keywords, labels, and various kinds of literals. This chapter tells how lexemes are written and classified in Owl, and discusses other syntactic matters.

The following notation will be used to describe Owl's syntax.

⇒	introduces an alternate expansion of a syntactic category
[*sequence-of-tokens*]	an optional *sequence-of-tokens*
syntactic-category ...	one or more repetitions of *syntactic-category*
syntactic-category ,...	one or more repetitions of *syntactic-category* separated by commas
syntactic-category ;...	one or more repetitions of *syntactic-category* separated by semicolons, optionally followed by a semicolon

Syntactic categories (like *expression*) are written in italics. Terminal symbols (like `operation`) are written in a typewriter font.

In Owl, commas are used to form lists of expressions (for example, argument lists), and are always separators. Semicolons are used to form lists of statements, and can be viewed as either separators or terminators.

Owl programs are written using *printing characters* (ASCII 41 octal through 176 octal) and *blank characters*.

printing-character ⇒

```
! " # $ % & ' ( ) * + , - . / : ; < = > ?
@ [ \ ] ^ _ ` { | } ~ 0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U
V W X Y Z a b c d e f g h i j k l m n o p
q r s t u v w x y z
```

blank-character ⇒ *space* *tab* *end-of-line*

Except for string literals (which can contain spaces), lexemes consist solely of printing characters. Blank characters are used to separate lexemes. One or more blank characters are required between adjacent identifiers or numeric literals. Blank characters are optional elsewhere.

Upper and lower case letters are equivalent except in string literals.

Comments are similar to lexemes, but are described in Section 3.7.

3.1 Identifiers

Identifiers (or id's) are used to name types, operations, exceptions, variables, objects, and so on.

An identifier consists of one or more alphanumeric characters. The first character must not be a digit.

id ⇒ *alphanumeric* ...

alphanumeric ⇒ *letter* *digit* _ # ?

letter ⇒ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

digit ⇒ 0 1 2 3 4 5 6 7 8 9

3.2 Reserved Words

Reserved words form a subset of the identifiers. These identifiers are used to form declarations and control flow expressions. They cannot be used to name objects, variables, operations, etc. The reserved words are listed in Appendix C.

3.3 Labels

Labels are used to name blocks of code. They serve as "targets" for `leave` expressions (see Section 5.13).

A label consists of an identifier enclosed in double angle brackets.

label ⇒ *<<id>>*

3.4 Integer Literals

An integer literal names an object of predefined type **Integer**.

An integer literal consists of one or more digits. It is always interpreted in base ten.

integer-literal ⇒ *digit ...*

3.5 Real Literals

Owl will eventually have a predefined type **Real**, and appropriate literals.

3.6 String Literals

A string literal names a constant object of predefined type **String**.

A string literal consists of zero or more printing characters or spaces enclosed in double-quote marks.

string-literal ⇒ *"printing-character-or-space ..."*

Double-quote, backslash, and the non-printing characters must be included via escape sequences. For now, the escape sequences are:

ESCAPE	MEANING
\"	"
\\	\
\n	*end-of-line*
\t	*horizontal tab*
ddd	*ASCII character ddd (octal)*

The octal code *ddd* can range from *000* to *377*. Further escape sequences will be specified in the future.

3.7 Comments

Comments in Owl begin with an exclamation point "!", and are terminated by the end of the current line. They can contain any legal character except end-of-line. Comments are semantically equivalent to spaces.

comment ⇒ ! *printing-char-or-space-or-tab ... end-of-line*

When comments are intended solely for the human reader, the exclamation point should be followed by a space. Characters other than a space may be used in the future to indicate that the comment is intended for programming environment tools in addition to the human reader.

Chapter 4

Simple Expressions

This chapter describes expressions in Owl. There are two broad catagories of expressions: simple expressions (the subject of this chapter), and control expressions (described in the next chapter). Simple expressions are executed to obtain a value or modify data. Control expressions are executed for their effect on the flow of control. Control expressions correspond to statements in many other languages.

Simple expressions come in the following forms.

expression	⇒	*literal*	
	⇒	*id*	
	⇒	*type-spec*	
	⇒	*invocation*	
	⇒	*unary-op-expression*	*cf*
	⇒	*binary-op-expression*	*cf*
	⇒	*record-constructor*	
	⇒	*sequence-constructor*	
	⇒	*simple-selection*	*cf*
	⇒	*indexed-selection*	*cf*
	⇒	*assignment*	
	⇒	*block*	

The expressions marked *cf* are called contracted forms. These are really alternative ways of writing more basic expressions (usually invocations). For example, `a + b` is really an abbreviation for `plus(a, b)`. Contracted forms are included in Owl to make programs more readable. We will explain each contracted form as we encounter it in the next two chapters. Appendix B gives a compact summary of all the contracted forms.

4.1 Value Required Contexts and Type Correctness

Owl is an expression language rather than a statement language. Everything that can be executed may be thought of as potentially having a value. However, this does not mean that every expression, or even every type of expression, actually has a value. In order to accomplish that, useless arbitrary values would have to be assigned to some expressions.

In some places, an expression is executed specifically to produce a value (for example, on the right hand side of an assignment, or as an argument to an invocation). Such places are called *value required* contexts. Other contexts do not require a value. Expressions that do not have values must not be used in value required contexts.

Every expression that has a value also has a *type*. This type expresses our compile time knowledge of the possible values that the expression can produce. The actual base type of the expression's value at run time will always be a subtype of the expression's compile time type. Whenever we refer to an expression's type, we are talking about our compile time knowledge. Whenever we refer to an object's type we mean its base type.

Associated with each value required context is a type requirement for that context. For example, the type requirement for the expression on the right of the ":=" in an assignment to a variable is simply the declared type of the variable. The type of an expression used in a value required context must be a subtype of the type requirement for the context in which the expression appears. This includes the case where the two types are equal.

Not all expressions, even in value required contexts, terminate normally. In particular, the escape expressions (**return**, **signal**, **leave**, etc.) cause some expression evaluation to be skipped. An expression is not required to conform to type requirements if it can be determined at compile time that the expression will be escaped from, and consequently, that its value will never be used. A common example of this is when all possible exits from a block being used as an operation body occur through **return** expressions: the type of the block (i.e. the type of its last expression) need not be a subtype of the **returns** type of the operation, and need not even have a value.

In the discussion that follows, unless some clarification is necessary, we will simply assume that expressions in value required contexts actually have values, and that the types of the expressions are correct for that context. We won't bother to mention these restrictions explicitly.

4.2 Order of Evaluation

Unless explicitly specified in the descriptions below, the order of evaluation of expressions is not defined. Therefore, the programmer should not depend on the side effects of one sub-expression for the proper evaluation of another. Even when the specification does allow this, it is generally considered a poor idea.

For example, in the expression `foo(x) + baz(x)`, if either `foo` or `baz` modify `x` in such a way as to affect the value of the other, the value of the expression is indeterminate, and may be different for different compilations.

4.3 Literals

A literal names a constant object of predefined type (currently just integers and strings). The form of literals is discussed in Chapter 3.

4.4 Identifiers

An identifier refers to the value of a local variable or fixed name, or it names a type. See, respectively, Sections 5.2, 6.7, and 6.1.

4.5 Type Specs

type-spec	⇒	*id*
	⇒	*id* `[` *expression* `,... ]`
	⇒	`Record [` *typed-id* `,... ]`
	⇒	`Fix_Rec [` *typed-id* `,... ]`
	⇒	*type-spec* `|` *type-spec*
	⇒	`subtype (` *type-spec* `)`
	⇒	`bind (` *id* `)`

A type spec refers to a type. Identifiers refer to simple (non-parameterized) types. An identifier followed by values in square brackets denotes a member of a type family. The `Record` and `Fix_Rec` forms denote members of particular predefined type families (see the Owl Library Manual).

`T | S` is the union of types `T` and `S`. It is also a type – the smallest type greater than (or equal to) both `T` and `S`. See Section 7.4 for a discussion of union types.

Both simple types and parameterized types are declared using type modules (the latter by way of type generators). Collectively, simple types and parameterized types are called *declared types.* As objects, declared types have base type `Declared_Type`, which is a subtype of type `Type`. Union types are not declared, and their base type is just `Type`.

The remaining two forms are not types in themselves, but are used to provide more precise control in defining generic operations or types. The use of these special type specs is described in Section 7.2.

4.6 Invocations

invocation ⇒ *operation-name* (*expression* ,...)

operation-name ⇒ [*type-spec* '] *id*

An invocation applies an operation to a *controlling object* along with a number of other argument objects. Execution proceeds as follows:

- All the argument expressions are evaluated.
- The operation name and the first argument (the controlling object) are used to locate the actual implementation.
- The implementation lists a number of argument variables and a body. The argument objects are assigned to the listed variables, and the body is executed.
- The value of the body, or an object designated by a **return** expression, becomes the value of the invocation expression.

The (compile time) type of the invocation is that given by the **returns** attribute of the operation definition. The base type of the returned object will always be a subtype of the type specified. The (compile time) types of the argument expressions must match those given in the operation definition.

To locate the actual implementation, we use the operation name and the controlling object. Since the operation name is almost always a simple identifier (no type qualification), we'll consider that case first.

For *instance* operations, the (compile time) type of the first argument, plus the operation name, is sufficient to locate the operation definition. To locate the actual implementation, we must take the following steps at run time:

- First, determine the base type of the controlling object. This is called the *controlling type*.
- Then, look up the named operation in the definition of the controlling type.

For *class* operations, we require that the first argument be a compile time known type. This type, plus the operation name, is sufficient to locate both the operation definition and the actual implementation.

The above rules are unambiguous as long as no class operation on type `T` has the same name as an instance operation on type `Declared_Type`. To avoid confusion, such a name conflict is forbidden.

The qualified form of operation name, `T'F`, refers to the operation `F` as defined and implemented in `T` rather than as defined and implemented locally. Qualified operation names are described in Section 6.6.3.

Restrictions:

If the first argument is a compile time known type (not a union), the operation name can denote a class operation defined on that type. Otherwise, the operation name must denote an instance operation defined on the type of the first argument.

If the arguments following the first one have types T_2 through T_n, and the operation definition lists additional argument types S_2 through S_m, then (1) n = m, and (2) each T_i must be a subtype of the corresponding S_i.

If the type of the first argument is T_1 | T_2 | ... T_n, then there must be an appropriate instance operation defined on each T_i. Each of these definitions is independently checked against the actual arguments. The `returns` types are unioned to get the type of the invocation.

If the operation named is an iterator, the invocation can only be used in the `in` clause of a `for` loop (see Section 5.7).

4.7 Unary Operation Expressions

The unary operation contracted forms are as follows:

`+` *A*	`u_plus (`*A*`)`
`-` *A*	`u_minus (`*A*`)`
`~` *A*	`u_tilde (`*A*`)`
`@` *A*	`u_at_sign (`*A*`)`
`not` *A*	`u_not (`*A*`)`

In each case, the expression form on the left is equivalent to the invocation on the right.

4.8 Binary Operation Expressions

The binary operation contracted forms are as follows:

A ^ *B*	caret (*A*, *B*)
A & *B*	ampersand (*A*, *B*)
A @ *B*	at_sign (*A*, *B*)
A | *B*	bar (*A*, *B*)
A * *B*	star (*A*, *B*)
A / *B*	slash (*A*, *B*)
A mod *B*	mod (*A*, *B*)
A div *B*	div (*A*, *B*)
A + *B*	plus (*A*, *B*)
A - *B*	minus (*A*, *B*)
A < *B*	less? (*A*, *B*)
A <= *B*	less_equal? (*A*, *B*)
A = *B*	equal? (*A*, *B*)
A >= *B*	greater_equal? (*A*, *B*)
A > *B*	greater? (*A*, *B*)
A |= *B*	bar_equal? (*A*, *B*)
A ~< *B*	not_less? (*A*, *B*)
A ~<= *B*	not_less_equal? (*A*, *B*)
A ~= *B*	not_equal? (*A*, *B*)
A ~>= *B*	not_greater_equal? (*A*, *B*)
A ~> *B*	not_greater? (*A*, *B*)
A ~|= *B*	not_bar_equal? (*A*, *B*)
A in *B*	in? (*A*, *B*)
A and *B*	and (*A*, *B*)
A cand *B*	(if *A* then *B* else false end if)
A or *B*	or (*A*, *B*)
A cor *B*	(if *A* then true else *B* end if)
A xor *B*	xor (*A*, *B*)
A eqv *B*	eqv (*A*, *B*)

In each case, the expression form on the left is equivalent to the invocation on the right. The precedence of these operators is discussed in Section 4.15 at the end of this chapter.

The and and or contracted forms evaluate both their arguments before the corresponding operations are invoked. The operators cand and cor provide for "short circuit" evaluation. This is useful when

- the validity of evaluating the second operand depend on the truth value of the first operand, i.e., `(B ~= 0) cand (A/B < 100)`, or

- unnecessary evaluation of the second argument may mean a significant performance penalty.

Note that the first argument of **cand** and **cor** must be of type **Boolean**. There is no way for a programmer to use the **cand** and **cor** operations with other types of first argument. The second argument to a **cand** or **cor** should also be of type **Boolean**. (This is not required, but disregarding this convention might lead to confusion.)

The contracted forms "~<", "~<=", "~>=", and "~>" have been added to allow precision when talking about types that are not completely ordered, i.e. types for which **not** A<B does not imply A>=B (e.g. a type representing sets). But for most types that define the comparison operators, **not_less**, **not_less_equal**, **not_greater_equal**, and **not_greater** should be made synonymous with **greater_equal**, **greater**, **less**, and **less_equal**.

4.9 Record Constructors

record-constructor ⇒ [*type-spec* '] { *record-field* ,... }

record-field ⇒ *id* : *expression*

A record constructor builds a new object from a group of named objects. Specifically,

```
{ a: x, b: y, c: z }
```

builds a fixed (i.e., constant) record containing **x**, **y**, and **z**, and

```
Complex'{ real: 3, imag: 25 }
```

might build the complex number 3 + 25i.

The (compile time) type of the expression {$\mathtt{id}_1$: $\mathtt{exp}_1$, ... $\mathtt{id}_n$: $\mathtt{exp}_n$} is

```
Fix_Rec [id1: T1, ... idn: Tn]
```

where T_i is the type of expression i. However, the object constructed at run-time has base type Fix_Rec [$\mathtt{id}_1$: B_1, ... $\mathtt{id}_n$: B_n] where B_i is the base type of the i^{th} value.

The expression

```
T'{ id1: e1, id2: e2, ... }
```

is translated to

```
construct ( T, {id1: e1, id2: e2, ...} )
```

and is intended to construct a new T object from the fixed record

```
{id1: e1, id2: e2, ...}
```

In a record constructor, the order in which the component expressions are listed is not significant.

See the Owl Library Manual for a full definition of fixed records.

Restrictions:

The type spec prefix must be a compile time constant expression.

4.10 Sequence Constructors

sequence-constructor ⇒ [*type-spec* '] { [*expression* ,...] }

A sequence constructor builds a new object from a list of other objects. Specifically,

```
{ x, y, z }
```

builds a fixed (i.e., constant) sequence containing x, y, and z, and

```
Set'{ x, y, z }
```

might build a Set containing x, y, and z.

The (compile type) type of the expression {exp_1, ... exp_n} is Fix_Seq[U] where U stands for the union of the types of the expressions within the braces. The base type of the object constructed at run-time is Fix_Seq[V] where V is the union of the base types of the values computed at run-time. If there are no expressions listed, U (and V) is the special union type Empty_Type.

The expression

```
T'{ e1, e2, ... }
```

is translated to

```
construct ( T, {e1, e2, ...} )
```

and is intended to construct a new T object from the fixed sequence {e_1, e_2, ...}. Note that the order in which the component expressions is listed can be significant.

See the Owl Library Manual for a full definition of fixed sequences.

Restrictions:

The type spec prefix must be a compile time constant expression.

4.11 Simple Component Selection

simple-selection ⇒ *expression* . *id*

The simple selection is a contracted form. When used as an expression,

```
P.C
```

is translated to

```
get_C (P)
```

The use of `P.C` as the target of an assignment is discussed in Section 4.13.2.

4.12 Indexed Component Selection

indexed-selection ⇒ *expression* [*expression* ,...]

The indexed selection is a contracted form. When used as an expression,

```
P [ e1, e2, ... ]
```

is translated to

```
get_indexed (P, e1, e2, ... )
```

The use of `P[e1,...]` as the target of an assignment is discussed in Section 4.13.3.

4.13 Assignments

assignment	⇒	*id* := *expression*	
	⇒	*simple-selection* := *expression*	*cf*
	⇒	*indexed-selection* := *expression*	*cf*

In Owl, an assignment is a value-returning expression. The value of an assignment to a variable (the first form) is just the value of the expression on the right hand side. In the case of the contracted forms this is a strongly encouraged convention, but is not enforced.

Syntactically, the assignment operator ":=" acts like an infix operator with minimum precedence and right associativity. Thus, `X := Y := A + B` is equivalent to `X := (Y := (A + B))` where `X` and `Y` are valid targets. (This assigns `A + B` to both `X` and `Y`.)

Note that, because of the low precedence of ":=" the expression, `A + X := B` is equivalent to `(A + X) := B`. Since `(A + X)` is not a valid target for an assignment, an error will be reported.

4.13.1 Variable Assignment

An assignment with a variable name as its target will set that variable to the value of the expression on the right of the ":=". This is the primitive form of assignment and is not equivalent to any operation or control expression.

Restrictions:

The target must be the name of a variable.

4.13.2 Simple Component Assignment

An assignment with a simple selection expression as its target is designed to invoke the `put` operation of a component. For example,

```
P.C := expr
```

is translated to

```
put_C (P, expr)
```

The result of the invocation becomes the value of the assignment expression.

4.13.3 Indexed Component Assignment

An assignment with an indexed selection expression as its target is designed to invoke the `put` operation of an indexed component. For example,

```
P [x, y, ...] := expr
```

is translated to

```
put_indexed (P, expr, x, y, ...)
```

Note the position of the assigned expression. Again, the result of the invocation becomes the value of the assignment.

4.14 Blocks

A block groups a sequence of expressions into a single expression. The simplest form of block is just an expression in parentheses. Thus, parentheses can be used to explicitly control the nesting of expressions. Blocks are fully defined in Section 5.12.

4.15 Precedence

The precedence of the various unary and binary operators is listed below. All binary operators are left associative except for "^" and ":=". The other expression forms are all "self delimiting," and do not need precedence rules.

HIGHEST PRECEDENCE

' *(in an operation name or constructor)*
A . id *A* [*B*,...]
+ *A* - *A* ~ *A* @ *A*
A ^ *B*
A & *B* *A* @ *B* *A* | *B*
A * *B* *A* / *B* *A* mod *B* *A* div *B*
A + *B* *A* - *B*
A < *B* *A* <= *B* *A* = *B* *A* >= *B* *A* > *B* *A* |= *B* *A* ~< *B* *A* ~<= *B* *A* ~= *B* *A* ~>= *B* *A* ~> *B* *A* ~|= *B* *A* in *B*
not *B*
A and *B* *A* cand *B*
A or *B* *A* cor *B*
A xor *B* *A* eqv *B*
A := *B*

LOWEST PRECEDENCE

Chapter 5

Control Expressions

This chapter describes the control expressions in Owl. These expressions are Owl's equivalent of statements.

Control expressions come in the following forms.

control-expression	⇒	*if-expression*
	⇒	*case-expression*
	⇒	*type-case-expression*
	⇒	*loop-expression*
	⇒	*for-expression*
	⇒	*exit-expression*
	⇒	*continue-expression*
	⇒	*return-expression*
	⇒	*yield-expression*
	⇒	*block*
	⇒	*leave-expression*
	⇒	*signal-expression*
	⇒	*resignal-expression*

These forms are discussed in the following sections, but first we introduce two additional forms: bodies and variable declarations.

5.1 Bodies

body	⇒	[*body-element* ;...]

body-element ⇒ *var-declaration*
⇒ *control-expression*
⇒ *expression*

A body is a sequence of control expressions, simple expressions, and variable declarations. These are to be evaluated in order. The value of a body is the value of its last element. An empty body has no value.

A body is always nested inside some other expression, which defines its extent. `Begin-end` brackets are never needed just to delimit nested code.

5.2 Variable Declarations

var-declaration ⇒ `var` *typed-id* [`:=` *expression*]

typed-id ⇒ *id* : *type-spec*

A variable declaration introduces a local name used to denote objects. Variables are given values by assignment (see Section 4.13.1). The type spec states that the variable will only denote objects of that type.

If the initialization clause "`:=` *expression*" is absent, the variable has no initial value and must be assigned to before it can be used.

If the initialization clause is present, it is equivalent to following the declaration with an explicit assignment.

A variable declaration can occur wherever a control expression can occur. The scope of a variable extends from the declaration itself to the end of the containing body.

A declaration without initialization has no value. A declaration with initialization takes its value from the initialization expression.

Restrictions:

The id in a variable declaration must not have been previously declared in the containing body. Although the scopes of variables can be nested (as in Algol), this is not recommended.

5.3 IF Expressions

if-expression ⇒ `if` *expression*
`then` *body*
[`elseif` *expression* `then` *body*] ...
[`else` *body*]
`end if`

The execution of an `if` expression proceeds as follows.

- The expression in the `if` clause is evaluated. If its value is `true` then the body in the `then` clause is executed, and evaluation of the conditional terminates. The value of the conditional in this case is the value of the body in the `then` clause.
- The expression following the first `elseif` (if any) is evaluated. If the result is `true` then the body in the `then` clause following the `elseif` is executed, and the evaluation of the conditional terminates. The value of the conditional in this case is the value of the body in the `then` clause.
- Each successive `elseif` clause is evaluated as above.
- If none of the `then` bodies have been executed (all of the boolean expressions evaluate to `false`), the body following the `else` is executed. The value of the conditional in this case is the value of the body in the `else` clause.

Restrictions:

The expressions following the `if` and `elseif` keywords must be of type `Boolean`. The `else` clause is not optional if the expression is in a value required context.

5.4 CASE Expressions

case-expression ⇒ `case` *expression*
case-arm ...
`end case`

case-arm ⇒ `on` *case-match* ,... `do` *body*
⇒ `otherwise do` *body*

case-match ⇒ *expression*
⇒ *expression* `to` *expression*

The expression immediately following the keyword `case` is evaluated. The resulting value (the *selector value*) is matched against each case-match in order. When a match is found, the corresponding case arm body (the body following the `do`) is executed, and the evaluation of the case expression terminates. The value of the `case` expression is the value of the case arm body executed.

The selector value is matched against a simple expression `e` by invoking the operation `equal?(`*selector_value*`, e)`. The selector value is matched against a range e_1 `to` e_2 by invoking the operation `between?(`*selector_value*`,` e_1`,` e_2`)`. The match succeeds if the invocation returns `true`, and fails if the operation returns `false`.

Only one `otherwise` arm is permitted, and it must be the last one. The body of the `otherwise` arm is executed when no case match is successful.

The special name `selector_value` can be used within the bodies of the case arms to refer to the value of the selector expression. The type of `selector_value` will be the (compile time) type of the selector expression.

Restrictions:

The implicit invocations of `equal?` and `between?` must be type correct, and must return an object of type `Boolean`. If `between?` is not defined for the selector expression, the range form of case match cannot be used. (All Owl types provide `equal?`.)

An `otherwise` arm is required if the expression is in a value required context.

5.5 TYPE_CASE Expressions

type-case-expression ⇒ `type_case` *expression*
type-case-arm ...
`end type_case`

type-case-arm ⇒ `on` *type-spec* ,... `do` *body*
⇒ `otherwise do` *body*

The `type_case` expression closely parallels the `case` expression. However, in a `type_case` expression, it's the base type of the selector value that is matched against the case arms, rather than the selector value itself.

The base type of the selector value is matched against a type spec `T` by invoking the operation `subtype?(`*selector_value*`.basetype, T)`. This effectively tests whether the selector value is of type `T`.

Only one `otherwise` arm is permitted, and it must be the last one. The body of the `otherwise` arm is executed when no case match is successful.

The special name `selector_value` can be used within the bodies of the case arms to refer to the value of the selector expression. Within a case arm of the form "`on` T_1,... T_n `do` *body*", the type of `selector_value` will be the union of T_1 through T_n. Within an `otherwise` arm, the type of `selector_value` will be the (compile time) type of the selector expression.

The special typing of `selector_value` is the whole purpose of including the `type_case` expression in Owl. A simple example should clarify the use of `type_case`. Assume that `Integer` is a subtype of `Real`, which is a subtype of `Complex`. The `add` operation for `Real` might be written as

```
operation add (me, addend: Complex)
    returns (Complex)
    is
    begin
        type_case addend
            on Integer do
                ! convert Integer addend to Real and do Real addition
                primitive_add (me, convert (mytype, selector_value));
            on Real do
                ! do Real addition
                primitive_add (me, selector_value)
            otherwise do
                ! the addend must be Complex - let him do it
                add (selector_value, me)
        end type_case;
    end;
```

Restrictions:

An `otherwise` arm is required if the expression is in a value required context, *and* the union of all the type specs does not cover the type of the selector expression.

5.6 LOOP Expressions

loop-expression ⇒ `loop`
body
`end loop`

The `loop` expression repeatedly executes its body until an `exit` expression is executed (see Section 5.8). An exception occuring in the body, or execution of a `leave`, `return`, or `signal`, can also terminate the loop.

Restrictions:

If a `loop` expression is used in a value required context, all `exit` expressions that apply to it must have an appropriately typed `with` clause.

5.7 FOR Expressions

for-expression ⇒ `for` *typed-id* `in` *invocation* `do`
body
`end for`

The invocation following the `in` keyword invokes an iterator, rather than a normal operation. When invoked, this iterator yields a series of objects one-by-one. The body of the `for` loop is executed once for each object in this series. The typed id introduces a local variable to denote the current object in the series.

The order of execution is as follows.

- Invoke the iterator, executing its implementation until it `yields` an object X.
- Assign X to the iteration variable.
- Execute the `for` loop's body.
- Resume executing the iterator's implementation just after the `yield`.
- Repeat these steps until
  - the iterator body terminates, or
  - control is forced out of the `for` loop's body, either by an exception or by an explicit `exit`, `leave`, `return`, `signal`, or `resignal`.

The scope of the iteration variable is the body of the `for` loop.

Even though execution of the `for` loop body and the iterator's body are interleaved, the scoping of local variables and the behavior of `leave`, `return`, and so on, is based on the local static structure of the program, not on the dynamic structure.

Restrictions:

The invocation must invoke an iterator. This iterator must yield objects assignable to the iteration variable. Since the loop body may not be executed at all, a `for` expression must not be used in a value required context.

5.8 EXIT Expressions

exit-expression ⇒ `exit` [`when` *expression*] [`with` *expression*]
⇒ `exit` [`unless` *expression*] [`with` *expression*]

An `exit` expression conditionally terminates the innermost `loop` or `for` expression containing it. The `when` or `unless` clause determines whether the exit is taken, and the `with` expression (if present) provides a value for the exited loop.

"`Unless` *expression*" is equivalent to "`when not` *expression*". If the `when` clause is missing, it is equivalent to "`when true`". If the `with` clause is missing, the exited loop has no value.

Restrictions:

An `exit` may only be used in the body of a repetition expression (a `loop` or a `for`).

The `when` or `until` expression must be of type `Boolean`.

The `with` clause must be used with any loop that is used in a value required context.

5.9 CONTINUE Expressions

continue-expression ⇒ `continue`

The `continue` expression causes termination of the body of the innermost containing repetition expression, but not of the repetition expression itself. The next cycle will thus be initiated.

Restrictions:

A `continue` expression may only be used in the body of a repetition expression (a `loop` or a `for`).

5.10 RETURN Expressions

return-expression ⇒ `return` [*expression*]

When a `return` expression is executed, the current operation invocation is terminated normally. The value of the optional expression is used as the value of the invocation.

Restrictions:

Return can only be used inside an operation implementation (including the get and put implementations for components). If the operation has a **returns** attribute, **return** must supply a value of that type. If the operation has no **returns** attribute, **return** must not supply a value.

5.11 YIELD Expressions

yield-expression ⇒ `yield` *expression*

The `yield` expression may be thought of as a special temporary **return** expression used in iterator implementations. The value of the sub-expression is passed back to the `for` expression that invoked the iterator, where it is assigned to the iteration variable. When the `for` loop body completes, execution continues following the `yield` expression.

Restrictions:

The `yield` expression may only be used in an iterator, i.e. an operation whose definition includes a `yields` attribute. `Yield` expressions have no value.

5.12 Blocks

block ⇒ `begin` [*label* ...]
body
[`except` *exception-handler* ...]
`end` [*label* ...]

⇒ ([*label* ...].
body
[`except` *exception-handler* ...]
) [*label* ...]

Without labels or exception handlers, a block simply delimits a body. The value of the block is the value of its body, i.e. the value of the last element in the body.

The labels at the beginning of the block serve as targets for **leave** expressions contained in the body. Note that it is not required that a **leave** expression referring to the label actually exist.

The labels at the end of the block must be a subset of those at the beginning. They serve as reader commentary and to aid error recovery.

The value of a labeled block is the value of its body unless a **leave** expression is executed, in which case the **leave** expression specifies the value.

A block can be used as either a simple or control expression. As a simple expression, it is usually just a grouping device. As a control expression, it can be used to define a body of code as a single action – either because we wish to **leave** the action, or because we want to catch exceptions arising within it.

Blocks delimited by "(...)" are usually used as simple expressions. Blocks delimited by **begin...end** are usually used as control expressions. Despite this convention, all blocks have the same semantics, however they are delimited, and however they are used.

Exception handlers are described in Section 5.15.

5.13 LEAVE Expressions

leave-expression ⇒ **leave** *label* [**with** *expression*]

The **leave** expression causes the containing block with the corresponding label to be terminated. The value of the block is the value of the expression following the keyword **with**.

Restrictions:

The **leave** expression may occur only within the lexical scope of a block having the same label as the one following the keyword **leave**. If that block is in a value required context the **with** clause must be used.

5.14 SIGNAL and RESIGNAL Expressions

signal-expression ⇒ **signal** *id* [**with** *expression*]

resignal-expression ⇒ **resignal**

When a "**signal E**" is executed, the current operation invocation is terminated, signalling the exception **E** at the call site. This exception will be caught by some handler in the body of the calling operation (see Section 5.15).

The expression following the keyword **with** is used as the value of the exception.

The **resignal** expression, which may appear only in an exception handler, causes the exception that was caught by that handler to be signaled from the current operation. The exception value is used even if the exception handler includes no **with** clause to name it.

`Resignal` is used to pass exceptions up the call chain.

Restrictions:

The signal id must be listed in the `signals` attribute of the operation containing the `signal` expression. If the `signals` attribute requires a value for the exception (by specifying a type for the value), then the `signal` must have a `with` clause. If no value is required, the `with` clause is not permitted.

A `resignal` expression may only appear in the body of an exception handler. `Resignal` is legal only if an explicit `signal` would be legal for each of the exceptions that the handler might catch.

5.15 Exception Handlers

exception-handler ⇒ `on` *id* ,... [`with` *typed-id*] `do` *body*
⇒ `otherwise` [`with` *typed-id*] `do` *body*

When an operation encounters an error or some other condition it cannot handle, it informs its caller of this fact by signalling an exception. It is the caller's responsibility to handle the situation (possibly by signalling *its* caller).

Exceptions are caught by exception handlers. Each handler is declared as part of a block, and catches exceptions arising within the main body of that block. The main body of the block is called the *scope* of the handler. A handler beginning "`on X, Y, Z`" catches exceptions `X`, `Y`, and `Z`. A handler beginning `otherwise` catches any exception.

If more than one handler could catch an exception, the handler that occurs first in the text actually catches it. This is because handlers attached to an inner block take precedence over handlers attached to a outer block, and the handlers in a single `except` clause are searched in text order.

When an exception is signalled, control is transferred to the body of the handler that catches that exception. When the handler body completes, execution continues immediately following the block containing that handler. The value of the handler's body (if any) becomes the value of the block.

Thus, each exception handler can be viewed as providing an alternate way to achieve the purpose of the block. The main body is the primary way, and the handler bodies are secondary ways that are used when exceptions occur.

Some exceptions have a value. (The presence of such a value, and its type, is listed in the `signals` attribute of the signalling operation.) To gain access to this object, the programmer can use a `with` clause. This introduces an identifier to denote the exception value. The scope of this identifier is the handler body. Also available within the handler

body is the special name **exception_name**, which denotes the name of the caught exception (as an uppercase string).

The exception **Failure** is used to indicate that an operation has failed in an unexpected manner. It frequently indicates a programming error. **Failure** is special in that it can be signalled by any operation whether listed in its **signals** attribute or not. **Failure** is always accompanied by a string argument explaining the cause of the failure.

To insure that there is always a handler to catch each exception, there is an implicit

```
begin
    . . .
except
    on Failure do resignal;
    otherwise do signal Failure
        with ("Unhandled exception: " & exception_name);
end;
```

surrounding the implementation of every operation.

Restrictions:

Let **E** be an exception that is caught by a handler with a **with** clause. We require two things:

- All operations that can signal **E** and are invoked within the scope of that handler must specify a value for **E**.
- The specified value must be assignable to the typed id declared by the **with** clause.

Handlers without **with** clauses place no restrictions on the body.

Blocks with exception handlers are occasionally nested. For example:

```
begin
    . . .
    begin
        . . .
        op (z);
        . . .
    except on Overflow do . . .
    end;
    . . .
except on Overflow with i: Integer do . . .
end;
```

An overflow exception signalled by `op(z)` will be caught by the inner handler, so the outer handler places no restrictions on this call.

Chapter 6

Types and Operations

This chapter describes how to define and implement new types in Owl. We first describe how to introduce an "isolated" type – one that does not inherit significant behavior from another type. Then we look at inheritance and the relation between a type and its supertypes.

Chapter 7 will show how to define families of related types.

6.1 Type Definitions

A type definition introduces a new type. The header of the definition lists a number of attributes of the new type, and the body defines all the operations and components provided for the type.

type-definition ⇒ `type_module` *id*
[*type-attribute* ...]
[*definition* ;...]
`end type_module`

type-attribute ⇒ `constant`
⇒ `mutable`
⇒ `base`
⇒ `abstract`
⇒ `subtype_of` (*type-spec* ,...)
[`when` (*expression*)]
⇒ `no_subtypes`
⇒ *note*

A type is *constant* if the individual objects of that type have no time-varying properties. A type is *mutable* if some of them do have properties that change. `Constant` and `mutable` are mutually exclusive attributes. These attributes document the programmer's intentions. Their accuracy cannot be guaranteed by the compiler.

A type `T` is a *base* type if there will be objects whose base type is `T`. A type `T` is an *abstract* type if all objects of type `T` actually belong to some subtype of `T`. `Base` and `abstract` are mutually exclusive attributes.

All operations defined for a base type must be implemented. Operations defined for an abstract type are not required to be implemented (although they may be). No allocation operations are allowed on abstract types (see Section 6.4).

The `subtype_of` attribute lists the immediate supertypes of the new type. (See Sections 2.2 and 6.6.1 for the implications of this.) Since all types are subtypes of `Object`, it is never necessary to list `Object` as a supertype. If multiple `subtype_of` attributes are present, all listed types are immediate supertypes.

The `no_subtypes` attribute can be used to prevent creation of any subtypes of the new type.

The body of a type definition contains a sequence of definitions. Within these, the keyword `mytype` serves as an abbreviation for the new type.

definition ⇒ *operation-definition*
⇒ *component-definition*
⇒ *inherit-definition*
⇒ *exclude-definition*
⇒ *fixed-name-definition*
⇒ *note*

Restrictions:

Each type must have a distinct id.

Each type spec listed in a `subtype_of` attribute must be a declared type. Unions, and the special forms `subtype(...)` and `bind(...)`, are not allowed.

6.2 Operation Definitions

An operation definition specifies (and possibly implements) a new operation for the enclosing type.

operation-definition ⇒ `operation` *id* `(` *control-spec* [`,` *typed-id* `,...`] `)`
[*operation-attribute* ...]
[*implementation*]

control-spec	⇒	`me`
	⇒	`mytype`
operation-attribute	⇒	`returns` (*type-spec*)
	⇒	`yields` (*type-spec*)
	⇒	`signals` (*exception* ,...)
	⇒	`override`
	⇒	*common-attribute*
exception	⇒	*id* [(*type-spec*)]

The first argument specifier (the *control spec*) is either `me` or `mytype`. `Me` is used for instance operations. Within an instance operation, `me` denotes the value of the first argument (the *controlling object*). `Mytype` is used for class operations. `Mytype` always denotes the type being defined by the enclosing type module.

The typed ids following the control spec introduce a number of local variables. These variables will be initialized with the actual argument objects when the operation is invoked. If the operation header lists N argument specifications, the operation must be invoked with N actual arguments. Each actual argument must be of the type listed in the corresponding specification.

The `returns(T)` attribute states that the operation returns a single object of type `T`. If no `returns` attribute is present, no object is returned.

The `yields(T)` attribute states that this operation is an iterator yielding objects of type `T`. `Yields` and `returns` are mutually exclusive.

The `signals(...)` attribute lists the exceptions that the operation can signal. No duplicate exception ids are allowed in this list. Each exception id can be followed by a type spec in parentheses. If present, the type spec states that an object of that type will be returned with the exception. The exception `Failure(String)` is implicitly part of every `signals` list. (`Failure` can also be listed explicitly, but its type must include `String`.)

Common attributes are described in Section 6.8.

The above information forms the *interface* of the operation (the part of the specification that is writable in Owl). The keyword `is` introduces the implementation. The implementation is optional, since (1) it is not needed when specifying (rather than implementing) a type, and (2) abstract types frequently have specifications for operations that are only implemented in subtypes.

There are four forms of operation implementation:

implementation	⇒	`is` *block*
	⇒	`is` *operation-name*

⇒ `is allocate` *block*
⇒ `is allocate` *operation-name*

The first form works as follows. When the operation is called, each argument object is assigned to the corresponding argument variable. Then the block is executed. If it terminates normally, the value of the block becomes the value of the call. `Return` or `signal` can be used to explicitly force termination.

The second form of implementation states that the operation being defined behaves exactly the same as another (existing) operation. "`Is` *operation-name*" is operationally equivalent to

```
is begin
    operation-name ( ... the arguments ... );    ! Pass all arguments through
    except otherwise do resignal;                ! Pass all exceptions back
    end
```

We say that the new operation is a *synonym* or *alias* for the existing operation.

The other forms of implementation are described in Section 6.4.

Restrictions:

Each instance operation of a type must have a distinct id, and each class operation of a type must have a distinct id. A class operation and an instance operation should use the same id only if they are closely related (see Section 6.4).

Unless an implementation block always returns or signals, it must produce a value of the specified return type.

An implementation by synonym is legal if the corresponding expansion is legal.

The scope of the argument variables is the entire operation definition.

Iterators can only be invoked as part of a `for` loop (see Section 5.7).

To avoid ambiguity, no class operation can have the same name as an instance operation on the type `Declared_Type` (see the Owl Library Manual).

6.3 Component Definitions

A component is a generalization of a record field or an array element. Components are thought of as having a value that can be read or written. The syntax is designed to capture this idea. For example, `X.count := Y.count` assigns the value of the count component of `Y` to the count component of `X`.

Like operations, there can be both instance and class components defined on type `T`. Instance components are part of the objects of type `T`. Class components are part of the type `T` itself.

In actuality, a component definition introduces two new operations: one to read the component value, the other to write it. These operations are usually invoked using contracted forms (see Sections 4.11 through 4.13).

component-definition ⇒ `component` *control-spec reference* : *type-spec*
[*component-attribute* ...]
[*get-or-put-definition* ...]
[*field-implementation*]

reference ⇒ . *id*
⇒ [*typed-id* ,...]

component-attribute ⇒ `get_only`
⇒ `put_only`
⇒ `override`
⇒ *common-attribute*

get-or-put-definition ⇒ *get-definition*
⇒ *put-definition*

get-definition ⇒ `get` [*operation-attribute* ...]
[*implementation*]

put-definition ⇒ `put` [*operation-attribute* ...]
[*implementation*]

field-implementation ⇒ `is field`

A simple component definition of the form

```
component control-spec . id : component-type
    component-attributes
```

is equivalent to the two operation definitions

```
operation get_id (control-spec)
    returns (component-type)
    component-attributes

operation put_id (control-spec, value: component-type)
    returns (component-type)
    component-attributes
```

Two attributes can modify this equivalence. The attribute **get_only** suppresses the creation of a **put_***id* operation. The attribute **put_only** suppresses the creation of a **get_***id* operation. These attributes are not "passed on" to the **put** and **get** operations.

Get_*id* and **put_***id* operations can be defined independently of component definitions. However, if a component definition **C** says **get_only**, then **put_C** cannot be defined independently for the current type. Similarly, **put_only** precludes an independent **get_C** operation.

A indexed component definition of the form

```
component control-spec [ typed-ids ] : component-type
    component-attributes
```

is equivalent to the two operation definitions

```
operation get_indexed (control-spec, typed-ids)
    returns (component-type)
    component-attributes

operation put_indexed (control-spec, value: component-type, typed-ids)
    returns (component-type)
    component-attributes
```

The **get_only** and **put_only** attributes behave as before.

The get-definition and put-definition clauses can be used to specify further attributes and an implementation for the **get** and **put** operations. These attributes are in addition to those listed as component attributes. (Note that a **returns** or **yields** attribute would conflict with the implicit **returns** attribute.)

As an alternative to giving specific implementations for **get_***id* and **put_***id*, the programmer can attach the clause **is field**. This states that the component is to be implemented as a field in the actual representation of the object (or type, if it's a class component). **Get_***id* will read this field, and **put_***id* will write it. A component implemented as a field is called a *field component*.

Restrictions:

This definition introduces operations named **get_***id* and **put_***id*. These names must not conflict with the names of other operations of the type.

The **get** and **put** operations must satisfy all restrictions stated in Section 6.2. **Get** and **put** implementations cannot be **is allocate**.

The **get_only** attribute precludes a **put** definition, the **put_only** attribute precludes a **get** definition, and a **field** implementation precludes both a **put** and **get** implementation. The **field** implementation is not permitted for the indexed form of component.

To avoid confusion with parameterized types (like **Array[Integer]**), indexed class components are not allowed.

6.4 Object Creation

Owl objects are created in two steps. First, storage is allocated to hold the object's field components. Then, each field must be initialized as needed. This gives rise to the `is allocate` form of operation implementation. The `allocate` keyword causes the storage to be allocated, and the block (or named operation) initializes the fields. The newly created object is then returned. The initialization block can use `me` to refer to the newly allocated object.

The `allocate` implementation can only be used for class operations that are defined to return an object of the controlling type. The base type of the created object is this controlling type.

Formally, an `allocate` operation is a pair of operations. The operation

```
operation id (mytype, i: T, ...)
    returns (mytype)
    operation-attributes
    is allocate block-or-operation-name
```

actually generates two operations. These are

```
operation id (mytype, i: T, ...)
    returns (mytype)
    operation-attributes
    is
    begin
        var a: mytype := ... a new (uninitialized) object ...;
        id (a, i, ...);
        return a;
    except otherwise do resignal;
    end

operation id (me, i: T, ...)
    allocate_visible
    operation-attributes
    is block-or-operation-name
```

If the *operation-attributes* include `public` (even by default), the instance operation is made `subtype_visible` instead (see Section 6.5).

Restrictions:

An `allocate` operation is not permitted for an abstract type `T`, since any `T` object really belongs to some subtype of `T`.

6.5 Visibility

To implement a type, implementations must be provided for its operations and components. In this process, it frequently happens that additional, implementation specific operations and components would be useful. In particular, additional field components are quite common.

To distinguish these "helper" definitions, the programmer should mark them `private`. Private operations and components are not visible outside the type definition. In fact, private instance operations can only be called with `me` as the first argument. (Private components can only be selected from `me`.) This insures that the base type of the object being manipulated is exactly `mytype`, and rather than some subtype of `mytype`.

Marking certain operations and fields as private allows the type implementer to retain control over exactly how the object being defined can be manipulated.

The attribute `public` is the opposite of `private`. `Public` is the default for operations and components of the type being defined.

The attributes `subtype_visible` and `allocate_visible` are alternate forms of `private`. `Subtype_visible` says that the definition is private, but that implementers of subtypes will be able to inherit or override the definition. (Definitions that are just `private` are not visible to subtypes.) `Allocate_visible` says that the definition is private, and it is further restricted to be used only in the `is allocate` form of operation implementation (or in other `allocate_visible` operations).

Since `public` contradicts `private`, it is incompatible with both `subtype_visible` and `allocate_visible`. However, any of the attributes `private`, `subtype_visible`, and `allocate_visible` can appear together.

visibility-attribute ⇒ `public`
⇒ `private`
⇒ `subtype_visible`
⇒ `allocate_visible`

6.6 Subtyping and Inheritance

The `subtype_of` attributes on a type definition `T` list the immediate supertypes of `T`. These types, along with their supertypes, their supertype's supertypes, and so on, are all *strict supertypes* of `T`. A type cannot be a strict supertype of itself. There is a single type `Object` that is a supertype of every other type. Hence, the absence of a `subtype_of` attribute is equivalent to one listing only `Object`.

The term *supertype* includes both the strict supertypes of `T` and `T` itself. We use *subtype* for the inverse relationship. `S` is a subtype of `T` whenever `T` is a supertype of `S`. `S` is a strict subtype of `T` whenever `T` is a strict supertype of `S`.

6.6.1 Subtype Compatibility

When S is declared to be a subtype of T, the programmer is asserting that every object of type S is also of type T. This means that every operation defined for T objects must be applicable to S objects as well. We require that an operation on S satisfy the same specification as the corresponding operation on T, or possibly a stronger one.

What do we mean by a stronger specification? An operation specification lists the arguments types that the operation must accept. The specification can be strengthened by requiring the operation to accept a larger range of arguments. An operation specification also lists the type of objects the operation is permitted to return. The specification can be strengthened by restricting the operation to a smaller range of return values.

This gives rise to the following rule:

The Subtype Compatibility Rule

Let S be a subtype of T. For every public instance operation F defined on T, there must be a corresponding public instance operation F defined on S. These two operations must be compatible in the following ways:

- F on S must take the same number of arguments as F on T.
- If F on T can take an argument of type A, then the corresponding argument for F on S must have type A (or some supertype of A).
- If F on S returns nothing, then F on T must return nothing.
- If F on S returns an object of type R, then F on T must return an object of type R (or some supertype of R).
- If F on S yields an object of type Y, then F on T must yield an object of type Y (or some supertype of Y).
- F on S can only signal exceptions already listed for F on T.
- If both operations signal E, and F on S specifies that E can have a value of type R, then F on T must specify that E can have a value of type R (or some supertype of R).

The subtype compatibility rule restricts the public instance operations, not the class operations or private instance operations. Only with public instance operations is there any uncertainty as to exactly which operation is being called (i.e., the base type of the controlling argument may not be known).

A component C is actually a pair of operations: get_C and put_C. These operations must both satisfy the subtype compatibility rule. The effect of this is that a component C on S must be specified to be a subtype of C on T if C on T is get-able; must be a supertype of C on T if C on T is put-able; and must be the same type as C on T if C on T has both put and get operations. Of course, this only applies if C on T is a public instance component.

6.6.2 Inheritance of Specifications

Typically, a subtype of T will have all the operations and components that T does, and a few more. It will be easier to understand the definition of a subtype S if the definition emphasizes the differences between S and T, rather than repeating all of T's operations and components. To facilitate this, we introduce the notion of *inheritance* – all of T's operations and components are automatically defined for S, unless the definition of S states otherwise.

The following paragraphs describe how inheritance works for operations. Let S be the type we are defining, and T be an immediate supertype of of S. Only public and subtype_visible operations of T are visible to S, so only these operations are inherited.

If F is an operation on T, then the specification of F is automatically added to the definition of S. This addition is done "textually" – names in the copied specification are interpreted in the context of S. For example, **mytype** now means S rather than T.

If S has more than one immediate supertype (as listed in the **subtype_of** attributes), it is possible that operation F is defined for more than one of these. This is legal if all the supertypes defining F agree on the interface of F. The interface includes: whether the operation is an instance or a class operation, the number and types of the arguments, and the **returns**, **yields**, and **signals** attributes.

If the supertypes do not agree, there are three alternatives. First, we can select a particular supertype to inherit from via the **inherit** definition. Second, we can choose to exclude F entirely via the **exclude** definition. Finally, we can write an explicit new specification for F on S. These alternatives are allowed even when there is no disagreement among the supertypes.

The **inherit** definition has the following form.

inherit-definition ⇒ **inherit** *id* ,... [**from** *type-spec*]
[**as** *visibility-attribute* ...]

When the **from** clause is present, it states that the operations listed are to be inherited from the given (immediate) supertype. The visibility attributes of a definition are normally inherited along with the specification. The **as** clause can be used to override this inheritance, or to explicitly state the desired visibility. In particular, the **as** clause can be used to resolve a disagreement among the supertypes.

The **exclude** definition has the following form.

exclude-definition ⇒ **exclude** *id* ,...

This states that the operations listed *do not exist* for the subtype. Of course, public instance operations cannot be excluded.

The third alternative is to write out a explicit new specification for F. If this new specification is different from one given in any of the immediate supertypes, this change must be indicated by the `override` attribute.

Components are inherited as two separate operations. These can be inherited, excluded, or respecified independently.

Restrictions:

The type specified in an `inherit` definition's `from` clause must be an immediate supertype of S. The identifiers listed in the `inherit` and `exclude` definitions must name operations provided on immediate supertypes. Since instance and class operations of the same name are expected to be intimately related, no mechanism is provided to inherit or exclude one but not the other.

No matter how a type's operations are defined, the type must satisfy the subtype compatibility rule.

6.6.3 Inheritance of Implementations

Operation implementations are inherited in much the same manner as specifications. If an operation F on S also exists (visibly) in one or more immediate supertypes, then the implementation of F can be inherited from these supertypes. To avoid ambiguity, we require that either there is exactly one supertype providing an implementation, or that all the supertypes providing implementations inherited their implementations from the same (common) supertype. In either case there is only one ultimate source of the implementation.

Implementation inheritance is basically "textual" – names in the copied implementation are interpreted in the context of S. This means that if the inherited implementation of F calls an operation G, then it will access the version of G appropriate for S, not T.

The effect of this rule is that a subtype programmer can modify the behavior of any inherited operation, and the other inherited operations will change correspondingly.

Recall that some of T's operations may not be visible to subtypes (specifically, private operations that are not subtype-visible). If an inherited implementation of F calls a non-visible operation, that operation is copied to S but with a different name. The new name is chosen so that it will never cause conflicts with existing or future operations. All uses of the renamed operation are also changed to use the new name. As a consequence, inherited operations that use non-visible operations will continue to work without the programmer having to be aware that such operations exist.

The renamed operations (called "secret" now) continue to be copied down the type hierarchy as long as they are needed.

It is fairly common for an operation F on S to be an extension of F on T. That is, it does everything F on T does, but also a little more. To facilitate this, S can use the notation T'F to denote the operation F as defined for type T. In effect, F is copied from T to S and renamed T'F. This operation is private to S. The notation T'F is called a *qualified name.*

If S has more than one immediate supertype, there may be more than one implementation available for F. This ambiguity can be resolved in two ways: an **inherit** definition can be used to select a particular specification and implementation pair to inherit, or we can provide an explicit new specification and implementation for F. These alternatives are allowed even when there is no disagreement among the supertypes.

The alias form of implementation can be particularly useful here. We can always select T's implementation for F by using **is** T'F as the implementation for F on S.

An explicit specification declared **override** never inherits an implementation.

As mentioned before, components are inherited as two separate operations.

When an object is created, space is allocated for all the object's fields. An object of base type S has all the fields explicitly declared for S, and all the fields declared by supertypes of S. There is one field for each **is field** implementation. If a field is inherited along more than one path, there is only one copy of the field in the actual representation.

Field inheritance is independent of the visibility of the component operations that access those fields. Fields that are not accessable may be optimized out.

Restrictions:

When a qualified name T'F is used within the definition of type S, T must be a supertype of S. (T is not restricted to the immediate supertypes, but must be a declared type.) F must be public or subtype_visible in T (otherwise the subtype cannot see it). There is no need for multiple qualification, as in U'T'F, so it is not permitted.

6.7 Fixed Names

A fixed name definition introduces a name for some object. The standard names for the booleans (**true** and **false**) are defined as fixed names. We call these names fixed because they always denote the same object.

fixed-name-definition ⇒ **define** *typed-id* [*common-attribute* ...]
:= *expression*

Fixed names are most often used to introduce local names for type specs or other constant objects. But they can also be used to denote mutable objects such as cannonicalization tables.

The value of fixed name is given by the "initialization expression". This expression is evaluated before the value of the fixed name is first used, but we do not specify exactly when. Because of this uncertainty, it is unwise for the initialization expression to have any (externally visible) side effects.

When the initialization expression is compile-time-known, the fixed name is also compile-time-known. This is why we can use fixed names in type specs.

Fixed names are private by default, but they can be given any visibility attribute.

Public fixed names are available outside the defining type module, and must not conflict with other public fixed names or the names of types. Public fixed names are not inherited, since this would cause an immediate name conflict.

Private fixed names are inherited. As with private operations, this is done "secretly" unless they were explicitly declared subtype-visible. Inherited fixed names have their initialization expression evaluated independently of their parent. Thus the two fixed names will denote the same object only if the expression re-evaluates to the same object.

Restrictions:

Type names and fixed names form a single name space. Outside any type module, each type and public fixed name must have a distinct id. Inside a type module, each fixed name (along with the externally defined types and public fixed names) must have a distinct id.

6.8 Other Attributes

All definitions have attributes. We have already seen the special attributes defined for types, operations, and components. There are a number of attributes common to several definitions. These are

common-attribute ⇒ *visibility-attribute*
⇒ **when** (*expression*)
⇒ *note*

The visibility attributes were explained in Section 6.5. The **when** attribute is described in Section 7.3. A note looks like

note ⇒ **note** (*note-entry* ,...)

note-entry ⇒ *id* [: *expression*]

Notes are a special form of comment directed at programming environment tools (see the Owl Programming Environment Manual for details). Each note entry is independent. The order of note entries, and their grouping into separate **note** clauses, is not significant.

Chapter 7

More on Types

The previous chapter discussed how to specify and implement simple types. In this chapter we discuss further features of Owl's type system, including

- Type Generators
- Special Type-Specs
- Conditional Subtypes
- Conditional Existence of Operations
- Union Types

7.1 Type Generators

A type generator is a form of type definition. A type generator defines a family of related types rather than a single type. For example, the `Set` type generator defines a set-of-integers type, `Set[Integer]`, a set-of-strings type, `Set[String]`, and so on. For any type `T`, the objects of type `Set[T]` are sets containing elements of type `T`.

Why do we need type generators? A single `Set_of_Anything` type would provide the same basic operations as `Set[T]`. The difference is in type checking. By using `Set[T]`, we can specify that we only want `T` objects to go into our set. The Owl compiler will then guarantee that we do not insert any other kind of object. Furthermore, we can assume that any object we extract will be of type `T`, so we can directly apply `T` operations to it. Without type generators, we would be forced to write a separate definition for each of the types `Set_of_Integers`, `Set_of_Strings`, and so on.

Simple types are named with a single identifier, like `Integer` or `String`. *Parameterized types* (members of a type family) are named by giving the generator name followed by one or more parameters in square brackets. `Matrix[Real]`, `Set[Matrix[Real]]`, and

`Table[Person,Address]` are all examples of parameterized types. `Matrix[Real]` is a type whose objects are real matrices, and `Table[Person,Address]` is a type whose objects map people to their home addresses.

Parameterized types can be used wherever simple types are used.

Two parameterized types are the same if they have the same generator and the same parameter values. If we declare

```
var x: Set [Integer];
var y: Set [Integer];
var z: Set [Real];
```

the variables `x` and `y` will have the same type. Different parameter values produce different types – so `y` and `z` have different types.

Currently, the parameters of a type generator must all be types. This limitation will be relaxed in the future.

Type generators make type specs more precise, and enhance our ability to do type checking. However, this can only work if the values of the type parameters are known at compile time. Consider the following (incorrect) Owl code.

```
var t: Type := ...;
var x: Table [String, t] := ...;
insert (x, "George", 37);
```

If `t` includes `Integer`, then the insertion is legal, otherwise it is not. Unfortunately, we cannot know this until the code actually executes. By allowing run time type parameters, we have lost the benefits of compile time checking.

In light of this, we require that the values of all type parameters be known at compile time. If we want the table `x` to contain objects of many different types, we should declare it to be `Table[String,Object]`.

A type generator definition is just a type definition with parameters:

type-generator ⇒ `type_module` *id* `[` *typed-id* ,... `]`
[*type-attribute* ...]
[*definition* ;...]
`end type_module`

Defining a type generator has the effect of defining an unbounded family of types – one type for each distinct combination of parameters. The type definition for a particular family member can be derived from the generator definition by substituting the actual parameter values for the parameter ids.

In effect, each of the definitions within the type module is replicated for each member of the family. In particular, when a class component is implemented as a field, there is a separate field for each member of the type family. Fixed names are not replicated (see Section 7.1.3).

Restrictions:

The typed ids (*id*: *type-spec*) declare the parameters of the generator. The scope of these parameters extends from the declaration itself to the end of the type module.

The parameter ids must all be distinct. The type specs must all be `Type` or some subtype of `Type`.

7.1.1 Using Parameters

Inside the definition of a type generator, the parameters act as named constants. They can be used anywhere a type value is needed – either in type specs or general expressions. As an example, consider the `Set` type generator:

```
type_module Set [Elementtype: Type]
operation insert (me, new: Elementtype)
    is
    begin
        var x: Elementtype;
        var y: Sequence [Elementtype];
        . . .
    end;
end type_module;
```

Here, `Elementtype` is used to specify the second argument to `insert`, and is also used to declare two local variables.

When we call `insert` on an object of type `Set[Integer]`, `Elementtype` will have the value `Integer` within the implementation of `insert`.

A type generator definition is correct if every legal combination of parameter values produces a correct (non-parameterized) type definition. In the `Set` example above, it would not be legal to assign an integer to `x`, because not all values of `Elementtype` will include `Integer`.

7.1.2 Subtyping

The `subtype_of` attributes in a type definition list the immediate supertypes of the type. By using type parameters, family members can be given different immediate supertypes. For example, the `subtype_of` attribute for `Set` could be

```
subtype_of (Collection [Elementtype])
```

which would make each `Set[T]` a subtype of the corresponding `Collection[T]`.

When a type `S` is declared to be a subtype of `T`, `S` can inherit both specifications and implementations from `T`. This is true for parameterized types as well as simple types. When `S` inherits a definition from a parameterized type like `Set[Integer]`, the parameter id `Elementtype` will be replaced by the actual parameter value `Integer`.

If a type module `A` inherits definitions from type module `B`, and `B` inherits definitions from `A`, confusion can arise about exactly which definitions are valid where. Fortunately, this situation does not seem to occur in practice. Inheritance cycles involving two or more type modules are not allowed in Owl.

For similar reasons, it is not legal to use a parameter id (like `Elementtype`) as an immediate supertype.

7.1.3 Fixed Names

Within a type generator, operation and component definitions are replicated for each member of the type family. Fixed names are not replicated. When we define a fixed name in a type generator, that fixed name is shared by all members of the type family.

This rule has two consequences. First, fixed names can be made public, since there is only one of them (no name conflicts). Second, fixed names can provide information shared among all the members of the type family.

Consider the following example.

```
type_module Table [Keytype: Type, Valuetype: Type]

define all_existing: Set [Object] private
    := create (Set [Object]);

component mytype.existing: Set [mytype]
    private
    is field;

operation create (mytype, ...)
    returns (mytype)
    is allocate
    begin
        . . .
        insert (mytype.existing, me);
        insert (all_existing, me);
    end;

end type_module;
```

Whenever a new object of type `Table[A,B]` is created, it is added to a set of all existing tables (`all_existing`) and to a set of all existing tables of its own type (`Table[A,B].existing`).

Restrictions:

Since fixed names are not replicated, it is illegal for any part of a fixed name definition to contain a parameter id or the keyword `mytype`.

7.2 Special Type-Specs

`Subtype(T)` is a special type spec used in declaring type parameters. The declaration `P: subtype(T)` says that `P` is a type *and* that `P` is a subtype of `T`. Let's define a type generator `Q` as follows.

```
type_module Q [X: subtype(Printable), Y: subtype(X)]
operation print (me)
    is
    begin
        var vx: X;
        var vy: Y;
        . . .
        vx := vy;
        . . .
    end;
end type_module;
```

The type `Q[A,B]` exists only if `A` is a subtype of `Printable` and `B` is a subtype of `A`. Within type module `Q`, `X` is known to be a subtype of `Printable`, and `Y` is known to be a subtype of `X`. This means that we can assign `vy` to `vx`, and apply `print` to both `X` and `Y` objects.

`Bind(S)` is a special type spec that allows us to write more precise operation interfaces. Let's assume that we want an operation that concatenates any two sequences. This can be specified as follows.

```
type_module Sequence [Elementtype: Type]
operation concatenate (me, x: Sequence [bind(S)])
    returns (Sequence [Elementtype | S]);
end type_module;
```

As a type spec, `bind(S)` stands for "any type". So the second argument to `concatenate` can be any sequence. If, for some particular invocation of `concatenate`, it turns out that

the actual type of the second argument is `Sequence[Integer]`, the identifier `S` is bound to `Integer` for the duration of the invocation.

The identifier `S` can be used as a type name throughout the definition of `concatenate`.

Consider the following code.

```
var x: Sequence [Character];
. . .
x := concatenate (x, "Henry");
```

`"Henry"` has base type `String`. `String` is a subtype of `Sequence[Character]`. So `S` binds to `Character`, and `concatenate` returns `Sequence[Character|Character]` which is `Sequence[Character]`.

Restrictions:

`Subtype(T)` can only be used as a parameter type spec in a type generator definition.

`Bind(S)` can occur only in an argument type spec in an operation or component definition. `Bind(S)` introduces a new identifier `S`. The scope of `S` extends from the bind to the end of the operation or component definition.

`Bind(S)` can be the entire argument type spec, or can occupy a type parameter position within the argument type spec. `Bind(S)` cannot occur within a union or `subtype(...)` form of type spec.

When a type spec containing a `bind(S)` is matched against the type of an actual argument, we apply the following algorithm to bind `S`:

- If `bind(S)` is matched against a type `B`, bind `S` to `B`.
- If `F[X]` is matched against a type `B`, match the type spec `X` against the smallest T such that `F[`T`]` is a supertype of `B`.

The subtype rules of Owl guarantee that the above algorithm gives a unique binding for `S`.

See the next section for another use of `bind`.

7.3 When Clauses

A `when` clause has the form

`when` (*expression*)

It can be attached to a **subtype_of** attribute, or to an operation, component, or fixed name definition. A **when** clause says that the attribute or definition is valid only when the expression is true.

Consider the following partial specification of the type **Printable**.

```
type_module Printable

operation print (me);
    ! print me on the standard output stream

end type_module;
```

Every printable object has a **print** operation. A set of objects is printable if the elements of the set are printable. That is, **Set[T]** is a subtype of **Printable** whenever **T** is a subtype of **Printable**. This sort of conditional subtyping can be expressed with a **when** clause. ("<=" is the subtype ordering on types.)

```
type_module Set [Elementtype: Type]
    subtype_of (Collection [Elementtype])
    subtype_of (Printable) when (Elementtype <= Printable)
```

If **Set[T]** is to be a subtype of **Printable**, it needs a **print** operation. This can be written

```
operation print (me)
    when (Elementtype <= Printable)
    ! print me on the standard output stream
    is
    begin
        var elem: Elementtype;
        . . .
            print (elem);
        . . .
    end;
```

The **when** clause says that the definition exists only for certain members of the **Set** type family – those that satisfy **Elementtype <= Printable**. Inside the definition of **print**, we can assume that **Elementtype** is a subtype of **Printable** (and thus has a **print** operation of its own).

If we want subtype relations within the **Set** type family, we can add another **subtype_of** attribute:

```
subtype_of (Set[bind(S)]) when (Elementtype < S)
```

This asserts that `Set[A]` is a subtype of `Set[B]` whenever `A` is a subtype of `B`.

As before, `bind(S)` stands for "any type". It introduces a local identifier `S` for use in the `when` clause.

Given actual parameter values, the compiler must be able to determine whether a `when` clause is true or not. To insure this, strict limits are placed on the form of the `when` expression. The `when` expression can refer to parameters of the containing type module, `bind` identifiers introduced in a `subtype_of` attribute, and literal types. The only operations permitted are `subtype?` on types (`<` and `<=`), and the two logical connectives `and` and `or`.

Restrictions:

`Bind(s)` can appear as a type parameter within a `subtype_of` attribute. `Bind(S)` introduces a new identifier `S`. The scope of `S` extends from the bind to the end of the containing type spec, and also through any attached `when` clause.

Type specs are matched against candidate supertypes using the matching rules described in the previous section.

A `subtype_of` attribute of the form

`subtype_of` (*T*$_1$, ... *T*$_n$) `when` (*condition*)

is equivalent to

`subtype_of` (*T*$_1$) `when` (*condition*)
. . .
`subtype_of` (*T*$_n$) `when` (*condition*)

in terms of semantics and name scoping.

Inheritance cycles involving two or more type modules are illegal. However, using conditional subtyping to relate members of the same type family is permitted.

Further restrictions should be placed on `when` expressions to insure that there is no ambiguity about subtyping. These will be specified at some future time. In any case, a type is a subtype of another type only if the `subtype_of` attributes *force* it to be a subtype.

7.4 Union Types

If `S` and `T` are two types, then `S | T` denotes the *union* of these two types. `S | T` is a type that includes the members of both type `S` and type `T`. The declaration

```
var x: S | T;
```

states that the variable x can be assigned either S objects or T objects as desired.

If an expression E is known to be of type S | T, then we can invoke any operation on E that is defined for both S and T objects.

There are no class operations for S | T.

S | T is a supertype of both S and T. If any type Q is a is a supertype of both S and T, then Q is a supertype of S | T. Thus, S | T is the "smallest" supertype of S and T.

As a consequence of the above definition, type union is associative, commutative, and satisfies the axiom

```
S <= T    ==>    (S | T) = T
```

Note that S | T is *not* intended to compute a least-upper-bound within the hierarchy of explicitly declared types. In particular, `Integer | String` is not a declared type.

There is also an "empty" union type, called **Empty_Type**. **Empty_Type** is a subtype of every other type. There are no objects of **Empty_Type** because they would have to have every behavior imaginable. Since **Empty_Type** has no objects, it needs no operations.

Appendix A

Syntax Summary

The following notation will be used to describe Owl's syntax.

⇒	introduces an alternate expansion of a syntactic category
[*sequence-of-tokens*]	an optional *sequence-of-tokens*
syntactic-category ...	one or more repetitions of *syntactic-category*
syntactic-category ,...	one or more repetitions of *syntactic-category* separated by commas
syntactic-category ;...	one or more repetitions of *syntactic-category* separated by semicolons, optionally followed by a semicolon

Syntactic categories (like *expression*) are written in italics. Terminal symbols (like `operation`) are written in a typewriter font.

In Owl, commas are used to form lists of expressions (for example, argument lists), and are always separators. Semicolons are used to form lists of statements, and can be viewed as either separators or terminators.

A.1 Lexemes

printing-character ⇒

```
! " # $ % & ' ( ) * + , - . / : ; < = > ?
@ [ \ ] ^ _ ` { | } ~ 0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U
V W X Y Z a b c d e f g h i j k l m n o p
q r s t u v w x y z
```

blank-character ⇒ *space* *tab* *end-of-line*

id ⇒ *alphanumeric* ...

alphanumeric ⇒ *letter* *digit* _ # ?

letter ⇒ A B C D E F G H I J K L M N O P Q R S T U
V W X Y Z a b c d e f g h i j k l m n o p
q r s t u v w x y z

digit ⇒ 0 1 2 3 4 5 6 7 8 9

label ⇒ <<*id*>>

integer-literal ⇒ *digit* ...

string-literal ⇒ "*printing-character-or-space* ..."

comment ⇒ ! *printing-char-or-space-or-tab* ... *end-of-line*

A.2 Simple Expressions

expression ⇒ *literal*
⇒ *id*
⇒ *type-spec*
⇒ *invocation*
⇒ *unary-op-expression* *cf*
⇒ *binary-op-expression* *cf*
⇒ *record-constructor*
⇒ *sequence-constructor*
⇒ *simple-selection* *cf*
⇒ *indexed-selection* *cf*
⇒ *assignment*
⇒ *block*

type-spec ⇒ *id*
⇒ *id* [*expression* ,...]
⇒ Record [*typed-id* ,...]
⇒ Fix_Rec [*typed-id* ,...]
⇒ *type-spec* | *type-spec*

	⇒	`subtype ( type-spec )`	
	⇒	`bind ( id )`	
invocation	⇒	*operation-name* `(` *expression* `,... )`	
operation-name	⇒	`[` *type-spec* `' ]` *id*	
record-constructor	⇒	`[` *type-spec* `' ] {` *record-field* `,... }`	
record-field	⇒	*id* `:` *expression*	
sequence-constructor	⇒	`[` *type-spec* `' ] { [` *expression* `,... ] }`	
simple-selection	⇒	*expression* `.` *id*	
indexed-selection	⇒	*expression* `[` *expression* `,... ]`	
assignment	⇒	*id* `:=` *expression*	
	⇒	*simple-selection* `:=` *expression*	*cf*
	⇒	*indexed-selection* `:=` *expression*	*cf*

A.3 Control Expressions

control-expression	⇒	*if-expression*	
	⇒	*case-expression*	
	⇒	*type-case-expression*	
	⇒	*loop-expression*	
	⇒	*for-expression*	
	⇒	*exit-expression*	
	⇒	*continue-expression*	
	⇒	*return-expression*	
	⇒	*yield-expression*	
	⇒	*block*	
	⇒	*leave-expression*	
	⇒	*signal-expression*	
	⇒	*resignal-expression*	
body	⇒	`[` *body-element* `;... ]`	

```
body-element          ⇒  var-declaration
                      ⇒  control-expression
                      ⇒  expression

var-declaration       ⇒  var typed-id [ := expression ]

typed-id              ⇒  id : type-spec

if-expression         ⇒  if expression
                         then body
                         [ elseif expression then body ] ...
                         [ else body ]
                         end if

case-expression       ⇒  case expression
                              case-arm ...
                         end case

case-arm              ⇒  on case-match ,... do body
                      ⇒  otherwise do body

case-match            ⇒  expression
                      ⇒  expression to expression

type-case-expression  ⇒  type_case expression
                              type-case-arm ...
                         end type_case

type-case-arm         ⇒  on type-spec ,... do body
                      ⇒  otherwise do body

loop-expression       ⇒  loop
                              body
                         end loop

for-expression        ⇒  for typed-id in invocation do
                              body
                         end for

exit-expression       ⇒  exit [ when expression ] [ with expression ]
                      ⇒  exit [ unless expression ] [ with expression ]
```

continue-expression ⇒ `continue`

return-expression ⇒ `return` [*expression*]

yield-expression ⇒ `yield` *expression*

block ⇒ `begin` [*label* ...]
body
[`except` *exception-handler* ...]
`end` [*label* ...]

⇒ `(` [*label* ...]
body
[`except` *exception-handler* ...]
`)` [*label* ...]

leave-expression ⇒ `leave` *label* [`with` *expression*]

signal-expression ⇒ `signal` *id* [`with` *expression*]

resignal-expression ⇒ `resignal`

exception-handler ⇒ `on` *id* ,... [`with` *typed-id*] `do` *body*
⇒ `otherwise` [`with` *typed-id*] `do` *body*

A.4 Definitions

type-definition ⇒ `type_module` *id*
[*type-attribute* ...]
[*definition* ;...]
`end type_module`

type-generator ⇒ `type_module` *id* `[` *typed-id* ,... `]`
[*type-attribute* ...]
[*definition* ;...]
`end type_module`

type-attribute ⇒ `constant`
⇒ `mutable`
⇒ `base`

```
                    ⇒   abstract
                    ⇒   subtype_of ( type-spec ,.... )
                              [ when ( expression ) ]
                    ⇒   no_subtypes
                    ⇒   note

definition          ⇒   operation-definition
                    ⇒   component-definition
                    ⇒   inherit-definition
                    ⇒   exclude-definition
                    ⇒   fixed-name-definition
                    ⇒   note

operation-definition ⇒  operation id ( control-spec [ , typed-id ,... ] )
                              [ operation-attribute ... ]
                              [ implementation ]

control-spec        ⇒   me
                    ⇒   mytype

operation-attribute ⇒   returns ( type-spec )
                    ⇒   yields ( type-spec )
                    ⇒   signals ( exception ,... )
                    ⇒   override
                    ⇒   common-attribute

exception           ⇒   id [ ( type-spec ) ]

implementation      ⇒   is block
                    ⇒   is operation-name
                    ⇒   is allocate block
                    ⇒   is allocate operation-name

component-definition ⇒  component control-spec reference : type-spec
                              [ component-attribute ... ]
                              [ get-or-put-definition ... ]
                              [ field-implementation ]

reference           ⇒   . id
                    ⇒   [ typed-id ,... ]

component-attribute ⇒   get_only
```

	⇒	`put_only`
	⇒	`override`
	⇒	*common-attribute*
get-or-put-definition	⇒	*get-definition*
	⇒	*put-definition*
get-definition	⇒	`get` [*operation-attribute* ...] [*implementation*]
put-definition	⇒	`put` [*operation-attribute* ...] [*implementation*]
field-implementation	⇒	`is field`
visibility-attribute	⇒	`public`
	⇒	`private`
	⇒	`subtype_visible`
	⇒	`allocate_visible`
inherit-definition	⇒	`inherit` *id* ,... [`from` *type-spec*] [`as` *visibility-attribute* ...]
exclude-definition	⇒	`exclude` *id* ,...
fixed-name-definition	⇒	`define` *typed-id* [*common-attribute* ...] `:=` *expression*
common-attribute	⇒	*visibility-attribute*
	⇒	`when (` *expression* `)`
	⇒	*note*
note	⇒	`note (` *note-entry* ,... `)`
note-entry	⇒	*id* [`:` *expression*]

Appendix B

Contracted Forms

Owl permits a number of alternate ways of invoking certain operations. These are called *contracted forms*. A contracted form is translated to its equivalent invocation as part of parsing. All of Owl's contracted forms are listed below. The contracted form is listed on the left and the equivalent invocation is listed on the right.

B.1 Unary Operators

`+` *A*	`u_plus` (*A*)
`-` *A*	`u_minus` (*A*)
`~` *A*	`u_tilde` (*A*)
`@` *A*	`u_at_sign` (*A*)
`not` *A*	`u_not` (*A*)

B.2 Binary Operators

A `^` *B*	`caret` (*A*, *B*)
A `&` *B*	`ampersand` (*A*, *B*)
A `@` *B*	`at_sign` (*A*, *B*)
A `|` *B*	`bar` (*A*, *B*)
A `*` *B*	`star` (*A*, *B*)
A `/` *B*	`slash` (*A*, *B*)
A `mod` *B*	`mod` (*A*, *B*)
A `div` *B*	`div` (*A*, *B*)
A `+` *B*	`plus` (*A*, *B*)
A `-` *B*	`minus` (*A*, *B*)

A < B	less? (A, B)
A <= B	less_equal? (A, B)
A = B	equal? (A, B)
A >= B	greater_equal? (A, B)
A > B	greater? (A, B)
A |= B	bar_equal? (A, B)
A ~< B	not_less? (A, B)
A ~<= B	not_less_equal? (A, B)
A ~= B	not_equal? (A, B)
A ~>= B	not_greater_equal? (A, B)
A ~> B	not_greater? (A, B)
A ~|= B	not_bar_equal? (A, B)
A in B	in? (A, B)
A and B	and (A, B)
A cand B	(if A then B else false end if)
A or B	or (A, B)
A cor B	(if A then true else B end if)
A xor B	xor (A, B)
A eqv B	eqv (A, B)

B.3 Components

A.C	get_C (A)
A.C := *expr*	put_C (A, *expr*)
A[e_1, e_2, ...]	get_indexed (A, e_1, e_2, ...)
A[e_1, e_2, ...] := *expr*	put_indexed (A, *expr*, e_1, e_2, ...)

B.4 Constructors

T'{id_1:e_1, id_2:e_2, ...}	construct (T, {id_1:e_1, id_2:e_2, ...})
T'{e_1, e_2, ...}	construct (T, {e_1, e_2, ...})

B.5 Precedence

The precedence of the various unary and binary operators is listed below. All binary operators are left associative except for "^" and ":=". The other expression forms are all "self delimiting," and do not need precedence rules.

HIGHEST PRECEDENCE

' *(in an operation name or constructor)*
A . id *A* [*B*,...]
+ *A* - *A* ~ *A* @ *A*
A ^ *B*
A & *B* *A* @ *B* *A* | *B*
A * *B* *A* / *B* *A* mod *B* *A* div *B*
A + *B* *A* - *B*
A < *B* *A* <= *B* *A* = *B* *A* >= *B* *A* > *B* *A* |= *B* *A* ~< *B* *A* ~<= *B* *A* ~= *B* *A* ~>= *B* *A* ~> *B* *A* ~|= *B* *A* in *B*
not *B*
A and *B* *A* cand *B*
A or *B* *A* cor *B*
A xor *B* *A* eqv *B*
A := *B*

LOWEST PRECEDENCE

Appendix C

Special Names

Owl has a number of special names that refer to commonly used objects. The special names frequently have different meanings in different contexts.

The special names currently in Owl include:

`me`	The controlling object. Used in operation and component definitions.
`mytype`	The controlling type. Used in operation, component, and fixed name definitions.
`value`	The value to be assigned. Used in **put** implementations for components.
`selector_value`	The value selected upon. Used in the arms of **case** and **type_case** expressions.
`exception_name`	The uppercase string name of the current exception. Used in exception handlers.
`Failure`	A special exception name. Implicitly present in every **signals** attribute, and implicitly generated by uncaught exceptions.

The following identifiers are used as binary operators and operator names.

```
and
cand
cor
div
eqv
in
mod
```

```
or
xor
```

The following identifiers are reserved. They cannot be used as names.

A unary operator:

```
not
```

Definition keywords:

```
allocate
as
bind
component
define
exclude
field
from
get
inherit
is
operation
put
type_module
var
```

Attributes:

```
abstract
allocate_visible
base
constant
get_only
mutable
note
no_subtypes
override
private
public
```

```
put_only
returns
signals
subtype_of
subtype_visible
when
yields
```

Expression keywords:

```
begin
case
continue
do
else
elseif
end
except
exit
fix_rec
for
if
leave
loop
on
otherwise
record
resignal
return
signal
subtype
then
to
type_case
unless
when
with
yield
```

Index

A

abstract types 9, 42
alias implementation for operations 44, 52
allocate_visible attribute 48
allocation operations 47
arguments
    subtype compatibility 49
    to operations 43, 44
    with bind() type specs 60
as clause in inherit definition 50
assignment 5, 6, 26
attributes
    common 43, 53
    list of 78
    of components 46
    of operations 43
    of types 41
    visibility 48

B

base types 9, 42
binary operators 21, 73
bind() 59
    in argument type specs 60
    in when() clause 62
blank characters 13
blocks 27, 36, 37
    with exception handlers 39
bodies 29

C

cand and cor 22
case expressions 31
class components 44, 46
class operations 7, 20, 47
commas 13, 65
comments 16
common attributes 43, 53
comparison operators 23
compatibility rule 49
components 11
    accessing values of 25
    assigning to 26, 44
    attributes of 46
    class 46
    contracted forms 25, 74
    defining 44
    field 46
    get_ and put_ operations 45
    get_ operations 25
    in type generators 57, 58
    indexed 25, 46
    inheriting 51
    initialization of fields 47
    instance vs. class 44
    public vs. private 48
    put_ operations 26
    selecting 25
    simple 25, 45
conditional existence of definitions 61
conditional subtyping with when() clause 61
constant objects 4
constant types 41
constructors
    contracted forms 74
    for records 23

for sequences 24
for user-defined types 23, 24
continue expression 35
contracted forms 11, 17, 21, 25, 73
control expressions 29
control spec in operation definition 43
controlling argument 4, 20
controlling object 20, 43
for private operations 48
controlling type 20
creating objects 47

D

declared types 19
definitions
conditional existence of 61
exclude 50
inherit 50
keywords 78
of components 44
of fixed names 52
of operations 42
of types 41
visibility of 48
when() clause 61

E

Empty_Type 24, 63
equality of types 55, 63
equivalence operators 23
escape expressions 18
escape sequences 15
exception handlers 36, 37, 38, 39
exception_name 38, 77
exceptions 5
catching 38
listing in operation header 43
unhandled 39
exclude definition 50
exit expression 33, 35
expressions 18
compile-time type vs. run-time type 18
control 29
keywords 79
order of evaluation 18
side effects 18
simple 17
values of 18
when union type 63

F

Failure exception 39, 43, 77
field components 8, 46
inherited 52
initializing 47
fixed names 11
defining 52
equivalence 53
in type generators 58, 59
inheritance of 53
initialization 52, 53
fixed records 23
fixed sequences 24
for expression 34
from clause
in inherit definition 50

G

get_ operations 25, 45
subtype compatibility 49
get_indexed operations 25, 46
get_only attribute 46

I

identifiers 14, 19
reserved 14, 78
if expressions 31
implementations
inheritance of 51
of components 46
of operations 43

of types 8
indexed components 46
assigning to 26
selection 25
infix operators 21, 73
inherit definition 50
inheritance 8, 9
cycles 58, 62
from parameterized types 58
of components 51
of fields 52
of implementations 51
of private definitions 51
of specifications 50
initialization
of fields 47
of fixed names 52
of objects 47
instance components 44
instance operations 7, 43, 49
integers 15
interface 7, 43
of inherited operations 50
invocations 20
compile-time type 20
contracted forms 11, 17
matching args with bind() 60
is allocate implementation 47
is field implementation 46
iterators 5
defining 43
invoking 21, 34
yielding from 36

L

labels 14, 36
leave expression 36, 37
lexemes 13
literals
integers 15
reals 15
strings 15
loop expression 33

M

me 43, 77
mutable objects 4
mutable types 41
mytype 42, 43, 77

N

name conflicts 20, 44, 46, 53, 78
names 14
in inherited code 50, 51
of supertype operations 20
parameter 57, 59
qualified 20, 51, 52
secret 51
special 77
type 10, 60
no_subtypes attribute 42
non-printing characters 15
note attribute 53

O

objects 3, 4
base type of 9
components 11
constant vs. mutable 4
creating 47
defining behavior of 3, 6
manipulating 7, 8
mutability 4
representing 8, 46
sharing 4, 6
type of 3, 6, 18
operations 3, 4
alias or synonym 44, 52
allocation 47
arguments to 43
attributes of 43
binary 21
class 7, 47
contracted forms 11, 17, 21, 73
defining 42

description 7
from supertypes 51
get_ and put_ 45
implementation forms 43
implementations required 42
in type generators 58
inheriting 9
inheriting implementations 51
inheriting specifications 50
instance 7
interface of 7, 43
invoking 20
locating 20
overriding 50
public instance subtyping 49
public vs. private 8, 48
returning 35
secret 51
specification of 42, 43, 49
subtype compatibility 49
synonym or alias 44, 52
unary 21
value of invocation 20, 35
operators
cand and cor 22
comparison 23
contracted forms 21
identifiers used as 77
precedence 27, 74
order of evaluation of expressions 18
otherwise
in case expression 32
in exception handler 38
in type_case expression 32, 33
override attribute 50, 52

P

parameterized types 19, 55
deriving definition of 56
equivalence 56
inheritance of 58
parameters 56
restricting with subtype() 59
using 57
when inheriting a definition 58
precedence of operators 27, 74
prefix operators 21, 73
printing characters 13
private attribute 48
private definitions 48, 51
public attribute 48
public instance operations 49
public vs. private 8, 48
put_ operations 26, 45
subtype compatibility 49
put_indexed operations 46
put_only attribute 46

Q

qualified names 20, 51, 52

R

reals 15
record constructors 23
reserved words 14, 78
resignal expression 37, 38
return expression 35
returns attribute 35, 43

S

secret fixed names 53
secret operations 51
selector value 32, 33
special name selector_value 32, 77
semicolons 13, 65
sequence constructors 24
sequences 24
sharing objects 4, 6
side effects of expressions 18
signal expression 37, 38
signalling 5, 38
signals attribute 38, 43
simple components
assigning to 26

selection 25
simple expressions 17
simple types 19
special names 77
specifications
inheriting 50
of operations 42, 43, 49
of types 7
overriding 50
subtype compatibility 49
vs. interfaces 7
statements 29
statements and expressions 11
strings 15
subtype compatibility 49
subtype() type spec 59, 60
subtype_of() attribute 42, 48, 62
with parameterized types 57
with when() clause 61
subtype_visible attribute 48
subtypes and supertypes 8, 48
subtyping
conditionally with when() clause 61
supertypes 48
accessing operations from subtype 51
declaring 42
multiple 50, 51
of parameterized types 57
union types as 63
synonym implementation for operations 44, 52
syntax notation 65

T

type checking 10
type generators 55
defining 56
fixed name sharing 58
parameters of 56, 57
type objects 7
type requirements
for expressions 18
for inherit from definition 51
for invocations 20, 21
for subtype compatibility 49
of parameters 56
type specs 10, 19
bind() 59
for variables 30
subtype() parameters 59
type_case expressions 32
type_modules 41
types 3, 6
abstract vs. base 9, 42
attributes of 41
compile-time vs. run-time 18
constant vs. mutable 41
declared 19
defining 41
equivalence 55, 63
examples 6
of signalled values 43
parameterized 19, 55
simple 19
specification and implementation 7
union 19, 62, 63

U

unary operators 21, 73
unhandled exceptions 39
union types 10, 19, 62, 63
in invocations 21
unless clause 35

V

value required contexts 18
value
special name 77
values
accompanying an exception 38, 39
of assignments 26
of control structures 11
of expressions 18
of fixed names 52
of invocations 20

variables 5, 6, 26
declaration of 30
for arguments 43, 44
initialization 30
iteration variable in for expression 34
of union type 62
scope of 30, 34
visibility 48
of secret operations 51
overriding inherited 50

W

when clause
in exit expression 35
when() clause for conditional definitions 60
expression part 62
on definitions 61
on subtype_of() 61
with clause
in exception handler 38, 39
in exit expression 35
in leave expression 37
in signal expression 37, 38

Y

yield expression 36
yielding 5, 34
yields attribute 43